MASTER EN SCIENCES MATHÉMATIQUES

Classification des êtres nommables dans les langages algorithmiques

Passau, Marie-Dominique

Award date:
1973

Awarding institution:
Universite de Namur

FACULTES UNIVERSITAIRES NOTRE-DAME DE LA PAIX - NAMUR

INSTITUT D'INFORMATIQUE

ANNEE ACADEMIQUE 1972-1973

Classification des Etres Nommables dans les Langages Algorithmiques

Marie-Dominique PASSAU

Mémoire présenté en vue de l'obtention
du grade de Licencié et Maître en Informatique

JURY DU MEMOIRE :

M. Cl. CHERTON

" What's in a name ?
that which we call a rose
by any other name
would smell as sweet. "

- Romeo and Juliette -
W. Shakespeare.

AVANT - PROPOS

Nous n'aurions pu entreprendre ni mener à bien ce travail sans l'aide compétente, les critiques, les encouragements et la confiance du Professeur Claude Cherton. Qu'il veuille trouver ici l'expression de notre plus vive reconnaissance.

Nous tenons à exprimer notre gratitude au Professeur J.N. Buxton pour l'intérêt qu'il a apporté à notre travail et pour ses réflexions critiques.

Nous sommes infiniment reconnaissant aux membres du "Department of Computer Science" et tout particulièrement au Professeur M. S. Paterson dont la gentillesse et le dévouement nous ont été précieux lors de notre séjour à l'Université de Warwick.

Enfin, nous voudrions remercier ceux qui ont bien voulu contribuer avec beaucoup de dévouement à la dactylographie de ce mémoire.

TABLE DES MATIERES

Chapitre Un

INTRODUCTION

A. Présentation du travail

Parmi les divers aspects des langages algorithmiques, nous nous proposons d'étudier le suivant : quels sont les êtres auxquels un langage donne ou permet de donner un nom ?

Pour les langages naturels comme pour les langages algorithmiques, parler de quelque chose implique que l'on définisse la chose en question au moyen de mots et de phrases existant déjà dans le langage et qu'ensuite on lui donne un nom. Il y a différentes façons de nommer une chose ou un objet. On peut l'appeler au moyen d'une périphrase ou d'une expression, ou aussi la nommer par un mot.

Exemple : "le petit garçon de la voisine" ou "Michel" peuvent être deux façons distinctes de désigner le même être.

Dans un langage algorithmique, l'action de nommer et de définir un être soit fait partie de la définition même du langage, soit s'effectue par une déclaration explicite ou implicite dans un quelconque programme écrit dans ce langage.

Exemple : * "3.14" est dit dans la définition d'Algol 60 être le nom du nombre réel représenté par 3.14 dans le système décimal.

* "integer x;" est une déclaration dans un programme Algol 60. Cette déclaration signifie : nous définissons un objet comme étant une variable à valeurs entières et nous l'appelons "x".

Il nous semble important de faire deux remarques à ce niveau :

- Nous nous trouvons en présence de trois concepts reliés entre eux : le nom, l'objet nommé et la représentation de l'objet.

On peut soit nommer un objet soit une représentation de cet objet. Par exemple, on peut très bien nommer une fonction et la représenter par une séquence d'instructions. Mais on peut aussi nommer la séquence d'instructionselle-

même. De même, le rapport de la circonférence au diamètre est un nombre réel que l'on peut appeler π et représenter par 3.14 par exemple. Mais il est aussi possible de dire que 3.14 est un"nombre réel", comme on le fait en Algol 60.
De plus, la représentation peut aussi être un nom (dénotation) de l'objet qu'elle représente (Algol 68).

- Les objets dont on parle, éventuellement par le biais de leur représentation, ont en général des propriétés. Aussi seront-ils très souvent groupés suivant leurs propriétés et leur nature de manière à former des modes ou types. La différence fondamentale entre un mode et l'ensemble de ses éléments est que le mode précise certaines propriétés de ses éléments et permet ainsi de pouvoir s'en servir. C'est pourquoi il existe des opérateurs définis sur la plupart des modes. Sous l'angle de la nommabilité, quatre possibilités sont envisageables : nommer un mode, des objets appartenant à ce mode, nommer des variables et des opérateurs sur ce mode.
Nous allons donc essayer de découvrir quels sont les objets et les modes dont les langages parlent. De plus, nous nous proposons de voir quels sont les objets et modes qu'il est possible de définir et de nommer par extension du langage et de quelle façon une telle extension peut se faire.

Nous nous intéresserons à titre exclusif aux êtres (objets et modes) dont le nom est formé d'un seul mot.
Nous nous pencherons aussi sur les liens qui existent entre la nommabilité et la définissabilité d'un objet. En effet, il est évident qu'on ne peut nommer un objet qui n'a pas été préalablement défini, que ce soit le langage ou le programmeur qui le définisse et/ou le nomme.
Dans ce but, nous étudierons d'abord la classification proposée par Ch. Strachey, nous analyserons ensuite celle que suggère Algol 68, et après comparaison et analyse critique, nous essayerons de classifier les êtres dont parlent les langages algorithmiques sur base des résultats obtenus.

B. Définitions

Tout au long de ce travail, nous emploierons les termes "nommé", "nommable", "défini" et "définissable" dans un sens différent de leur signification habituelle dans la langue française. C'est pourquoi nous voulons attirer l'attention sur ce fait et donner les définitions dont nous nous sommes servis.

"nommé" : se dit d'un objet auquel le langage considéré a donné un nom.

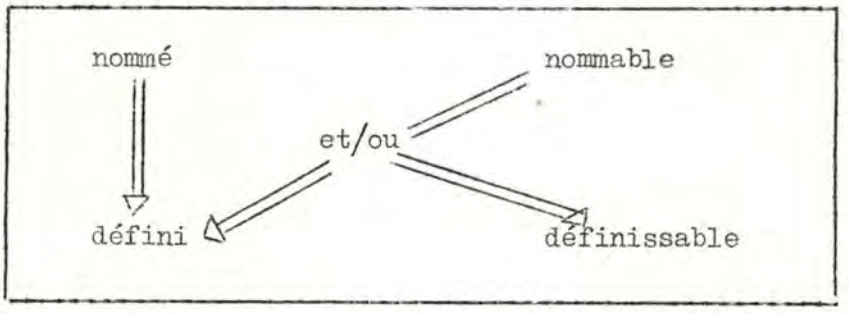
"défini" : se dit d'un objet dont le langage auquel il appartient a donné une définition.

"nommable" : se dit d'un objet auquel le programmeur peut affecter un nom dans le langage considéré.

"définissable" : se dit d'un objet que le programmeur peut définir dans le langage considéré.

La seule et unique exception à ce principe a été faite dans le titre, où "nommable" y a le sens courant et donc signifie "nommé" et/ou "nommable" par rapport aux définitions données ci-dessus.

Il existe des relations entre ces quatre propriétés. C'est ainsi que tout objet nommé est défini, que tout objet nommable est défini et/ou définissable (ou non exclusif), mais que l'inverse n'est pas vrai. Le schéma ci-dessous résume la situation:



Chapitre Deux

LA CLASSIFICATION DE STRACHEY

A. Aperçus de la Théorie de Strachey

Dès 1966, on trouve dans la littérature scientifique des articles de Ch. Strachey, professeur à Oxford, sur un aspect particulier des langages de programmation : l'assignation. Il l'analyse, la décortique et cette étude a profondément marqué la suite de ses recherches sur les langages. C'est ainsi qu'en 1971 et 1972, il publie, seul ou en collaboration avec D. Scott, des articles à propos de mathématisation de la sémantique des langages algorithmiques, et entre autres la classification qui nous intéresse ici.

1. Mécanisme de l'assignation

Les langages de programmation, suivant en cela l'exemple des langages naturels et de la mathématique, permettent à l'utilisateur de désigner les objets par des noms. La relation entre l'objet désigné et son nom est une fonction au sens mathématique du terme, que Strachey appelle environnement. Nous définirons maintenant quelques termes que Strachey utilise continuellement dans la suite.
Il appelle Id le domaine des noms ou identificateurs. L'ensemble des objets qui peuvent être nommés est D (D pour "denotation", c'est-à-dire "chose signifiée, dénotée ou désignée" par un mot, par opposition au sens que ce mot a en Algol 68).

"En mathématique, l'objet désigné par un nom reste constant pendant toute l'existence lexicographique du nom. Cette propriété n'est vérifiée, en langage de programmation, que pour certains objets tels que des procédures. Par contre, pour des noms d'objets de type réel, entier, booléen, il est possible de changer la valeur, c'est-à-dire l'objet associé au nom, par une assignation. La valeur associée à des noms de ce genre ne peut donc être obtenue que dynamiquement et en fonction de l'état courant de la mémoire" nous explique Strachey.

Il est à remarquer que cet avis porte à discussion. En effet, on peut très bien considérer que ce qui est associé au nom n'est pas la valeur réelle, entière ou booléenne, mais un objet à valeurs dans $\mathbb{R}$, $\mathbb{N}$ ou $\{t,f\}$, ce que l'on appelle couramment une variable. Il y a donc une ambiguïté, que Strachey est amené à lever de la façon suivante.
Il considère que le nom désigne une "location" (*) qui reste fixe et que la valeur associée au nom est le contenu de cette location. Il parle alors de "$\mathcal{L}$-valeur" du nom pour la location et de "$\mathcal{R}$-valeur" du nom pour le contenu de celle-ci. ($\mathcal{L}$ pour "left" et $\mathcal{R}$ pour "right"). Ces deux objets ne sont donc pas indépendants, bien que de nature différente.

Une location représente une zone de mémoire quelle qu'en soit la taille. Ce qui nous intéresse à leur sujet sont les deux propriétés suivantes : elles peuvent contenir des objets et sont nommables (addressables comme on l'exprime généralement).

La mémoire se partitionne en un nombre fini de locations. On appellera état de la machine à un moment donné, ou plus exactement état de sa mémoire, le rapport entre l'ensemble des locations et leur contenu à ce moment.

Soit S l'ensemble de ces états, S sera donc défini comme suit :

$$S = [L \to V]$$

où L est l'ensemble des locations,
V l'ensemble des valeurs que l'on peut stocker dans la mémoire.

$X \to Y$ représente une fonction quelconque appliquant X dans Y, X et Y étant deux ensembles quelconques,

et $[x]$ l'ensemble des fonctions x.

Ayant maintenant S, L et V à notre disposition, définissons les fonctions qui nous permettrons d'extraire et de modifier le contenu d'une location.

(*) "location" : ce mot est pris au sens anglais du terme. Etant donné que nous n'aurons jamais besoin du mot français location, nous continuerons dans la suite à utiliser le terme anglais.

Soit $\alpha, \alpha' \in L$, $\beta \in V$ et $\sigma, \sigma' \in S = [L \to V]$

Ces fonctions sont les suivantes :

$$\text{Contents} : \quad L \longrightarrow [S \to V]$$

$$\text{et Update} : \quad L \times V \longrightarrow [S \to V] \qquad (*)$$

Donc $\quad \text{Contents}\,(\alpha)(\sigma) = \sigma(\alpha)$

et si $\sigma' = \text{Update}\,(\alpha, \beta)(\sigma)$ alors

$$\text{Contents}\,(\alpha)(\sigma') = \beta$$

$$\text{et} \quad \text{Contents}\,(\alpha')(\sigma') = \text{Contents}\,(\alpha')(\sigma) \text{ si } \alpha \neq \alpha'$$

ce qui correspond à première vue à notre idée intuitive de mise à jour. Avant de pouvoir examiner l'assignation, nous avons encore besoin de fonctions qui fournissent la valeur des expressions. Puisque les expressions, dont les noms sont un cas particulier, ont à la fois des $\mathcal{L}$ - et $\mathcal{R}$-valeurs, il nous faut deux fonctions, soit $\mathcal{L}$ et $\mathcal{R}$.

Remarquons que la valeur d'une <u>expression</u> ne dépend pas uniquement de l'état courant de la mémoire, soit σ, mais aussi de l'environnement, soit ρ, qui précisera les objets désignés par les identificateurs figurant dans l'expression.

Nous aurons donc :

$$\mathcal{L} : \text{Exp} \longrightarrow [\text{Env} \longrightarrow [S \to L \times S]]$$

$$\mathcal{R} : \text{Exp} \longrightarrow [\text{Env} \longrightarrow [S \to V \times S]]$$

où Exp est l'ensemble des expressions possibles dans le langage
et $\text{Env} = [\text{Id} \longrightarrow D]$ l'ensemble des environnements.

Le S intervenant dans le résultat est dû aux éventuels "effets de bord" lors de l'évaluation.

(*) Pour des raisons de concordance de notations avec Strachey, nous emploierons les termes anglais partout où il n'y aura pas ambiguïté, c'est-à-dire principalement pour les noms de fonctions et de domaines. Lors d'un premier emploi, ils seront généralement entourés de guillemets.

L'instruction d'assignation sous sa forme la plus générale s'écrit:

$\mathcal{E}_0 := \mathcal{E}_1$ où $\mathcal{E}_0$ et $\mathcal{E}_1$ appartiennent à Exp. Quel en est le mécanisme ?

Remarque : la plupart des langages ne permettent d'avoir comme expressions en partie gauche que celles constituées d'un seul identificateur.

L'opération s'exécute en trois étapes :

1. Trouver la $\mathcal{L}$-valeur de $\mathcal{E}_0$.
2. Trouver la $\mathcal{R}$-valeur de $\mathcal{E}_1$.
3. Faire la mise à jour.

(on a supposé une exécution de gauche à droite).

Si on part de l'état initial σ_0 et que l'on travaille dans l'environnement ρ, cela donne :

1. $\mathcal{L}(\mathcal{E}_0)(\rho)(\sigma_0) = \langle \alpha, \sigma_1 \rangle$

   où α est la $\mathcal{L}$-valeur de $\mathcal{E}_0$ et $\sigma_1 = \sigma_0$ s'il n'y a pas eu d'effets de bord.
2. $\mathcal{R}(\mathcal{E}_1)(\rho)(\sigma_1) = \langle \beta, \sigma_2 \rangle$

   où β est la $\mathcal{R}$-valeur de $\mathcal{E}_1$.
3. Update $(\alpha, \beta)(\sigma_2) = \sigma_3$.

L'effet de l'assignation est donc de passer de l'état σ_0 à l'état σ_3.

2. Classification

Strachey différentie les domaines, ou ensembles d'objets, en domaines de base et domaines dérivés, de manière à pouvoir détailler D, V et Exp. Comme lui, nous prendrons les exemples dans Algol 60.

2.1. Domaines de Base

Intuitivement, les domaines de base sont ceux que le programmeur n'a pas la possibilité de définir lui-même.
Si l'on veut préciser la définition intuitive donnée par Strachey, on arrive à la définition suivante qui est nettement plus rigoureuse :

> La condition nécessaire et suffisante pour qu'un domaine soit de base est que ses éléments soient définis et non définissables.

Cette définition sera étendue aux modes au chapitre cinq et la démonstration sera faite à ce moment-là.

Les domaines de base les plus fréquemment rencontrés sont R (réels), N (entiers), T (booléens), C (caractères) et Q (chaînes de caractères). Ce sont les "données" les plus importantes, c'est-à-dire les objets que l'on traite. Ces domaines sont fixés par le langage qui définit les noms standards de ses éléments et le nom standard du mode correspondant si celui-ci existe.

Exemple : real est le nom du mode réel

-1.15_{10} est le nom d'un réel particulier.

A ceux-là viennent s'ajouter les "points de branchement" J (jump points), les locations L (vu ce qui a été analysé au paragraphe précédent) et S.

Si l'on examine ces différents domaines sous l'angle de la nommabilité et en considérant aussi bien les éléments du domaine que le mode ou type correspondant s'il existe, on aboutit au tableau T.II.A.1.
Précisons que ce tableau s'applique à Algol 60 et que le terme "spécifié" s'entend "nommé par spécification au sens d'algol 60", ce qui veut dire que le nom standard du domaine n'est utilisé que pour spécifier au compilateur le type d'un paramètre formel dans une procédure.

Domaines de base	Eléments	exemples	Mode correspondant
R	nommés	-1.15_{10} "var"	∃ nommé : real
N	nommés	12	∃ nommé : integer
T	nommés	false "var"	∃ nommé : boolean
Q	nommés	'ab␣cd'	∄ spécifié : string
J	nommables		∄ spécifié : label
L	nommables		
S	/		∄ / /

T. II.A.1

2.2. Domaines composés ou dérivés

Nous allons maintenant voir quels sont les domaines que le programmeur peut définir lui-même et de quelle façon il peut le faire.

Les domaines dérivés sont "construits" à partir d'autres domaines déjà définis (donc soit de base, soit dérivés), au moyen des outils d'extension fournis par le langage lui-même. Puisque les domaines sont des ensembles, les outils employés sont tout simplement des opérations ensemblistes dont voici les plus utilisées : soient X_o et X_1 deux domaines quelconques, on peut construire leur somme $X_o + X_1$ (union disjonctive) et leur produit cartésien $X_o \times X_1$. Cela s'applique bien sûr à plus de deux domaines. On posera, pour simplifier les notations, :

$$X^n = X \times X \times \ldots \times X \qquad n \text{ facteurs}$$

et $X^* = X^0 + X^1 + X^2 + \ldots$

Exemples : I + R est le domaine des nombres entiers ou réels.

R x R couples de réels, peut être une représentation des complexes.

Une troisième méthode pour composer deux domaines est d'établir des fonctions à arguments dans l'un et valeurs dans l'autre.
Pour des raisons qui paraîtront évidentes dans le paragraphe suivant, S ne sera utilisé comme "matériau" de base qu'au moyen de cette troisième méthode.

Quels sont, selon Strachey, les domaines dérivés que l'on peut construire en Algol 60 ?
Appelons D l'ensemble des objets nommables. On a :

- les $\mathcal{R}$-valeurs possibles des expressions :

$E = \{$ valeurs des expressions $\} \neq Expr = \{$Expressions$\}$.

$E = D + V$.

E contient D puisque les noms sont des cas particuliers des expressions, donc les valeurs associées à un nom sont toutes susceptibles d'être associées à une expression.

- les procédures sans type et avec type :

$$P = \left[D^* \longrightarrow [S \to S]\right] + \left[D^* \longrightarrow [S \longrightarrow V \times S]\right]$$

D^* représente une liste de longueur quelconque ($D^* = D^0 + D^1 + D^2 + \ldots$) d'arguments qui sont nommables puisque représentés par des paramètres formels.

Le corps de la procédure est une "commande", donc elle modifie l'état de la machine.
Si c'est une procédure avec type (fonction), le résultat appartient à V puisqu'il faut l'assigner au nom de la procédure dans le corps de celle-ci, d'où la présence de V x S dans le second membre.

- <u>les tableaux</u> : L^{**}

On peut assigner des valeurs aux éléments d'un tableau, ce sont donc des locations.

Les vecteurs : $A_1 = L + L^2 + L^3 + \ldots$ longueur quelconque
$= L^*$

Les matrices : $A_2 = A_1^1$ matrices (n,1)
$+ A_1^2$ matrices (n,2)
$+ A_1^3$ matrices (n,3)
$+ \ldots$ ...
n quelconque.
$= (L^*)^*$

Les tableaux à trois dimensions $A_3 = (A_2)^* = ((L^*)^*)^*$

Donc les tableaux à un nombre quelconque de dimensions :
$A = L^* + (L^*)^* + ((L^*)^*)^* + \ldots$ que nous conviendrons avec Strachey de noter L^{**}.

- <u>les "calls by name"</u>, c'est-à-dire les paramètres formels appelés par nom; ce sont en fait, au point de vue sémantique, des procédures avec type et sans paramètres. Elles produisent une valeur qui n'est pas restreinte à V mais peut être n'importe où dans E + Q, et peuvent avoir un"effet de bord".

$$W = [S \longrightarrow (E + Q) \times S]$$

W pour "wriggly values", c'est-à-dire des objets dont la valeur est différente à chaque fois qu'on les regarde.

Nous pouvons maintenant construire D, l'ensemble des choses nommables.

Dans D, on trouvera d'abord les domaines de base dont les éléments sont nommables, donc J et L, puis les domaines dérivés, E non compris. En effet, E est formé de l'union de deux ensembles: celui des objets nommables D et celui des objets nommés, V. (Bien que ce ne soit pas la définition première de V, un simple coup d'oeil au tableau T.II.A.2 en montre la validité - Q ne joue quasi aucun rôle en Algol 60, on ne se formalisera donc pas de son absence dans V). Pour Algol 60, on a donc :

D = L locations, donc variables réelles, entières, booléennes et paramètres formels appelés par valeur

+ J "jump points"

+ L^* tableaux et paramètres formels de type away appelés par valeur

+ W calls by name

+ P procédures

+ W_1^* switches

où P et W ont été définis plus haut et W_1 est donné par :

$$W_1 = [S \longrightarrow E_d \times S] \qquad E_d = \{\text{$\mathfrak{Z}$-valeurs possibles des expressions de désignation}\} = J + W_1$$

En fait, D est un domaine dérivé mais qui n'est ni définissable ni nommable dans le langage tout comme V. Ce sont donc plutôt des domaines que l'on pourrait, en accord avec Strachey, appeler caractéristiques d'un langage.

$V = T + R + N$ (Algol 60)

L'intérêt de considérer D, V et E réside dans la distinction suivante faite par Strachey :

"Certains objets peuvent apparaître dans des expressions, être assignés à une variable, ou encore apparaître comme paramètres actuels dans un appel de procédure. (Il s'agit principalement des nombres). Puisque, nous allons le voir de suite, ces objets sont privilégiés par rapport à d'autres, nous les cataloguerons en une classe "première". Une procédure, par contre, ne peut apparaître que dans un appel de procédure soit comme opérateur soit

comme paramètre actuel. Il n'y a pas d'autres expressions mettant des procédures en jeu ou dont le résultat est une procédure. Donc, en quelque sorte, les procédures sont des citoyens de seconde classe en Algol 60, c'est-à-dire des objets qui doivent toujours apparaître en personne et ne peuvent jamais être représentés par une variable ou une expression - sauf dans le cas de paramètres formels où l'objet de seconde classe, que ce soit une procédure, un "string" ou une étiquette, sera désigné par un nom. Si ce nom désigne en fait un quelconque objet du domaine (J, P ou Q), ce n'est pas une variable à proprement parler car on ne peut lui assigner de valeur."

Il est à remarquer que lorsque Strachey parle de domaine, c'est la plupart du temps et de façon implicite, des éléments de ce domaine qu'il s'agit. En fait, Strachey se soucie apparemment peu de savoir si un domaine est nommable ou s'il lui correspond un mode.

Etendons maintenant le tableau T.II.A.1 aux domaines dérivés et ajoutons-y la classe de ces différents domaines; nous obtenons ainsi le tableau T.II.A.2. Dans ce tableau, nous avons utilisé les abréviations suivantes :

AS pour "éléments assignables"
MEV pour "membres possibles d'une expression" (en tant que constante ou variable).
B pour "de Base"
et Dér. pour "dérivé".

On peut y remarquer que Q n'appartient ni à D ni à V. Cela découle du rôle quasi nul joué par les strings en Algol 60. En effet, ils ne servent que pour transmettre des paramètres à des procédures écrites dans un autre langage ou code. Il faut se rendre compte que ce qui est nommable en Algol 60 n'est pas un string mais un paramètre formel de type string; le paramètre actuel correspondant ne pourra pas être appelé par valeur (Revised Report : 4.7.5.1 et 4.7.5.4) et il y aura donc remplacement littéral du nom du paramètre formel par le string lui-même (c'est-à-dire sa dénotation dans le langage puisque les strings sont nommés).

1. Domaines	2. Base / Dérivé	3. $\in$ D	4. $\in$ V	5. Eléments définition nom.exemple	6. Mode correspondant ∃ définition nom.	7. Classe AS. MEV.n°
R	B	non	oui	définis nommés -1.15_{10}	∃ défini nommé: real	oui oui 1 (1)
N	B	non	oui	définis nommés 12	∃ défini nommé: integer	oui oui 1 (1)
T	B	non	oui	définis nommés true	∃ défini nommé: boolean	oui oui 1 (1)
Q	B	non	non	définis nommés 'abx␣d'	∄ — spécifié: string	non non 2
J	B	oui	non	définis nommables	∄ — spécifié: label	non oui 2 (5)
L	B	oui	non	définis nommables	∄ — —	oui oui 1 (2) (2)
L*	Dér.	oui	non	définissables nommables	∃ défini nommé: array	oui oui 1 (3) (3)
P	Dér.	oui	non	définissables nommables	∃ défini nommé: procedure	non non 2 (4) (4)
W_1*	Dér.	oui	non	définissables nommables	∃ défini nommé: switch	non oui 2 (5)
W	Dér.	oui	non	définissables nommables	∄ — —	non non 2

ALGOL 60 T.II.A.2. (1) à (5) voir commentaires.

On s'étonnera peut-être de trouver "défini" en colonne six pour L*, P et W_1*. Ces domaines sont bien dérivés en ce sens que les éléments en sont définissables. Si toutefois le mode correspondant est défini et nommé, cela est dû à la définition du langage et n'affecte en rien la dérivabilité des éléments. C'est précisément là que réside l'outil d'extensibilité d'Algol 60 qui définit le mode et laisse à l'utilisateur le soin de définir les éléments qu'il a envie d'utiliser. Le mode est défini de façon assez souple pour que l'utilisateur ait encore quelque liberté dans son choix des éléments.

La dernière colonne (7) nécessite un certain nombre de commentaires qui se rattachent tous à la remarque suivante :

Strachey, en établissant sa distinction entre première et seconde classe d'objets, s'est inspiré des propriétés de ces objets par rapport à l'assignation. Sont-ils oui ou non assignables, peuvent-ils faire partie d'une expression ? Cependant, après avoir pris tant de soin à définir les $\mathcal{L}$- et $\mathcal{R}$-valeurs, il examine les objets tantôt sous leur $\mathcal{L}$-aspect, tantôt sous leur $\mathcal{R}$-aspect pour les classifier. Expliquons-nous : (les chiffres entre parenthèses se rapportent aux indices correspondants dans la colonne 7 du tableau T.II.A.2.).

(1) R, N et T sont assignables, c'est évident. Il s'agit ici de $\mathcal{R}$-valeurs, d'objets appartenant à V.

(2) Par contre, les locations en tant que telles, en Algol 60, ne sont pas assignables. En effet, L $\notin$ V! Mais la $\mathcal{R}$-valeur qui leur est associée, c'est-à-dire leur contenu, l'est.
Quand un nom figure en partie droite d'une assignation, on l'évaluera en $\mathcal{R}$-mode. On considère donc dans ce cas-ci que le nom désigne non pas une location mais un réel, un entier ou un booléen. Mais le nom en fait, nous dit Strachey, désigne une location et la $\mathcal{R}$-valeur contenue dans celle-ci est associée au nom mais non pas désignée directement par lui; donc les locations sont assignables par le biais de leur contenu.

(3) Les tableaux, en réalité, ne sont ni assignables, ni membres possibles d'une expression, mais bien leurs éléments qui sont des locations. Strachey admet cependant que les tableaux sont de première classe.

(4) Que les procédures ne soient pas assignables, et qu'elles ne puissent faire partie d'une expression, voilà qui est surprenant. Bien sûr, seules les applications de fonctions le sont (Revised Report : 3.3.1), et cela ne suffit pas pour généraliser, mais n'aurait-on pas dû distinguer les deux cas. Selon Strachey, les procédures sont néanmoins de seconde classe parce qu'elles ne sont ni assignables ni membres d'expression et doivent toujours apparaître en personne. Il ne considère donc plus ici la $\mathcal{R}$-valeur d'une procédure, quand elle existe, mais bien la procédure elle-même.

(5) Si les labels et switches peuvent être membres d'une expression, il s'agit uniquement d'expressions de désignation (Revised Report : 3.5.1), expressions particulières qui ne sont utilisées que dans les "goto statements". Strachey les considère malgré tout comme des objets de seconde classe.

En réalité, le fait pour un objet d'être ou ne pas être assignable n'est pas dû à lui-même, à l'existence ou l'absence d'une $\mathcal{R}$-valeur associée mais bien à la volonté du ou des auteur(s) du langage de permettre ou d'interdire une pareille possibilité.

Il nous semble que la clarté de la classification nécessiterait une définition plus rigoureuse des conditions d'appartenance aux deux classes ainsi que de l'assignabilité. La façon de procéder indiquée par Strachey laisse en effet trop de place à l'arbitraire et nous croyons que cela masque des aspects, qui peuvent être importants, des langages ainsi analysés.

On trouvera en annexe I la classification complète d'Algol 60 selon Strachey.

B. Relations nom-objet-valeur

1. Conséquences du mécanisme de l'assignation

Reprenons les domaines qui jouent un rôle important dans le mécanisme de l'assignation décrit au paragraphe A.1. Nous avons Id, les noms, L, les locations et V, les objets stockables. Les fonctions qui les relient ont déjà été décrites :

$S = [L \to V]$ les états de la mémoire,

$$\mathcal{L} : Exp \longrightarrow \Big[Env \longrightarrow [S \longrightarrow L \times S]\Big]$$

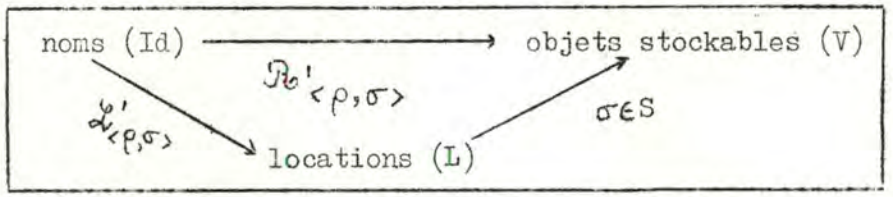
et $$\mathcal{R} : Exp \longrightarrow \Big[Env \longrightarrow [S \longrightarrow V \times S]\Big] .$$

Vu que le cas le plus fréquent, et généralement le seul permis, est d'avoir un identificateur unique en partie gauche d'une assignation, nous poserons :

$\mathcal{L}'$: Id x Env x S $\longrightarrow$ L x S

et $\mathcal{R}'$: Id x Env x S $\longrightarrow$ V x S

On se convaincra sans peine de ce que, pour $\rho \in$ Env et $\sigma \in$ S fixés et Exp réduit à Id, $\mathcal{L}'$ et $\mathcal{R}'$ sont équivalents à $\mathcal{L}$ et $\mathcal{R}$ respectivement. On obtient ainsi le diagramme II.B.1,



II.B.1.

où $\mathcal{R}'_{<\rho,\sigma>}(x) = \mathcal{R}'(x,\rho,\sigma) = \mathcal{R}(x)(\rho)(\sigma)$

et $\mathcal{L}'_{<\rho,\sigma>}(x) = \mathcal{L}'(x,\rho,\sigma) = \mathcal{L}(x)(\rho)(\sigma)$

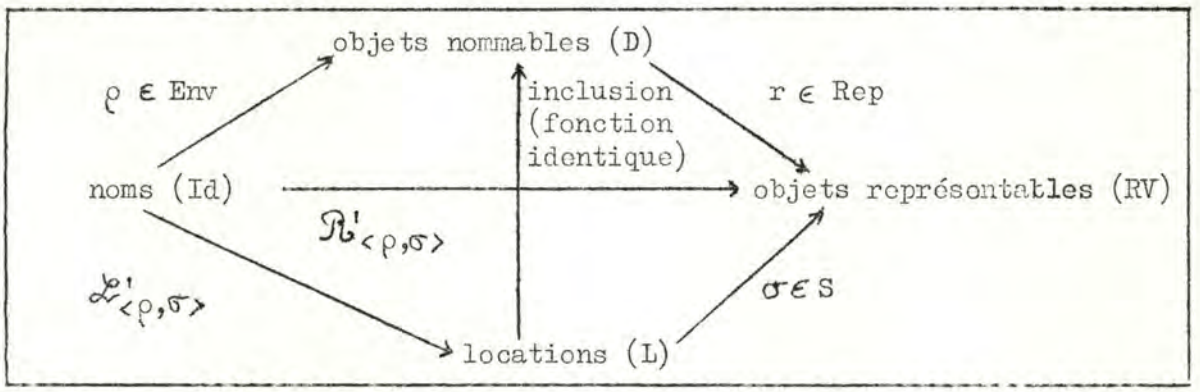
avec $x \in$ Id.

Cependant, ce qui nous intéresse principalement, ce n'est pas tellement Id, L ou V, mais bien D.
Nous allons donc essayer de compléter le diagramme. Quelles sont les relations entre Id, L, V et D ?
Id est relié à D par Env, les environnements; L est une partie de D; mais pour V, il y a un problème : en effet, si l'on considère uniquement Algol 60 et les descriptions de D et V faites par Strachey, on voit que ces deux domaines sont totalement disjoints. Mais faut-il en conclure qu'un objet nommable n'est pas stockable en mémoire ? Non bien sûr, sinon où serait l'utilité de nommer des objets ?
Il faut remarquer que ce ne sont pas les objets eux-mêmes qui sont stockés mais bien une représentation de leur $\mathcal{R}$-valeur. Il faudra donc que les objets dérivés que l'on nomme soient définis en termes d'objets ayant une $\mathcal{R}$-valeur. C'est la raison principale qui nous a poussé à introduire Rep un peu plus loin. De plus, dans certains langages, on sera amené soit à définir la $\mathcal{R}$-valeur de certains objets comme étant une location proprement dite, soit à convenir d'une représentation des locations en termes de $\mathcal{R}$-valeur de manière à pouvoir les stocker. C'est cette deuxième solution

qui a été adoptée en BCPL comme nous le verrons au paragraphe C.1., tandis que la première est surtout d'application en Algol 68 (voir chapitre III). De plus, nous allons envisager maintenant les langages dans leur ensemble et non plus uniquement Algol 60. Or, parmi ceux-ci, il en est qui permettent de nommer des objets qui sont déjà nommés dans la définition du langage, donc des objets peuvent appartenir à la fois à D et à V.

Si, dans le diagramme II.B.1, nous remplaçons V par RV, le domaine des objets représentables en mémoire, ce diagramme n'est en rien modifié quant à sa signification puisque $RV \supseteq V$. En effet, des objets comme les "jump points" ne sont pas stockables au sens strict du terme, c'est-à-dire susceptibles d'être contenus dans une location, mais on peut quand même les représenter en mémoire par un objet du genre "adresse".

Nous allons maintenant définir un ensemble de fonctions $Rep = \{ r : D \longrightarrow RV \}$. Ce sont des "représentations". En fait, r associera à une chose nommable sa représentation en termes de choses définies et donc représentables en mémoire. Il va de soi que r varie avec le langage considéré, et c'est en raison de cela que nous avons défini l'ensemble de ces fonctions. Si l'on applique r à une location, ce qui est possible puisque les locations sont nommables, on obtiendra un objet représentable, "address-like", correspondant à la location. Cet objet appartient en fait à $RV \setminus V$. Par contre, l'application de σ à une location en donne le contenu, c'est-à-dire un objet stockable appartenant à V. Donc $\sigma(L) \cap r(L) = \emptyset$. On obtient ainsi le diagramme II.B.2.



II.B.2.

2. Analyse critique du diagramme II.B.2.

A l'exception de Rep, nous n'avons fait que reprendre les définitions et les fonctions données par Strachey pour construire ce diagramme. Or, nous savons que, en fonction des définitions de Env et des $\mathcal{L}$-valeurs, à un nom correspond un seul objet et une seule location. Nous nous proposons d'examiner tout d'abord la partie gauche du diagramme, c'est-à-dire les "flèches" partant de Id.

2.1. Nom-objet

Puisque à un nom correspond un et un seul objet, et réciproquement, $\rho \in$ Env est une fonction bijective et la relation entre un nom et l'objet qu'il désigne est purement statique (pour un programme donné).
Avant d'aller plus loin, on pourrait se demander s'il n'est pas possible qu'à un même nom correspondent plusieurs objets ni que plusieurs noms désignent le même objet ?

a) Une propriété importante des identificateurs va nous aider à voir plus clair : il s'agit du "range" ou encore "scope". Dans la plupart des langages, les déclarations des "identificateurs", même implicites, n'ont qu'une validité limitée, généralement au bloc dans lequel elle est définie pour les langages à structure de bloc. Il est donc fort possible et même fréquent qu'un même identificateur désigne des objets distincts dans des blocs ou des "ranges" distincts.

Dans un même "range", maintenant, il est fort possible qu'un même nom désigne plusieurs objets, le contexte seul précisant lequel est à prendre en considération.

Exemples : - "1" peut être un "label" ou un nombre entier (Pascal, Fortran, Algol 60,...)
- les mécanismes d'adressage indirect (par pointeur) et de références successives (Algol 68) peuvent, pour un même nom situé en partie droite d'une assignation, donner un objet ou un autre suivant le type de l'objet dont le nom figure en partie gauche (coercition)
- "+" désigne un opérateur arithmétique entier ou réel suivant le contexte.

b) Inversément, dans certains langages, mais pas dans tous, il est permis de déclarer des "synonymes" et dans ce cas, à un même objet peuvent correspondre plusieurs noms.

Exemple : - let p = q (C.P.L.)

- int a = 3 ; int b = a (Algol 68)

En conclusion, nous pouvons dire que Env n'est pas un ensemble de fonctions mais plutôt le graphe d'une relation "désigne" entre Id x Ra et D où Ra est l'ensemble des "ranges". Donc Env = (Id x Ra) x D = $\{((x,i),0) \mid x \in Id, i \in Ra, 0 \in D$ et x est un nom de $0\}$ (deux triplets sont distincts s'ils diffèrent pour une des trois composantes au moins).

ρ sera donc un sous-ensemble de Env.

En réalité, considérer Env comme un graphe ne résout à peu près le problème que si la seule ambiguïté possible quant à l'objet désigné par un nom est due au range. Si, de plus, le contexte doit intervenir pour décider quel est l'objet ainsi désigné, Env = (Id x Ra) x D n'est pas suffisant. On voit que, sauf pour les cas simples d'un langage à structure de bloc où des relations "un nom-plusieurs objets" dans un même range sont impossibles, il est très difficile sinon quasi impossible de formaliser exactement Env.

2.2. Nom-location

Nous venons de voir qu'en pratique à un même nom ne correspond pas toujours un même objet et réciproquement. Pour des raisons semblables, à un même nom peuvent correspondre plusieurs locations et vice versa.

a) En effet, puisqu'un nom est susceptible de désigner plusieurs objets, il l'est aussi de désigner plusieurs locations, celles dont le contenu sera l'objet stockable correspondant. De plus, dans le cas de procédures récursives par exemple, des objets tels que des variables locales seront désignés par un même nom mais il y correspondra des locations différentes.

b) Inversément, si un objet peut être dénoté par un ensemble de synonymes, une location sera donc éventuellement la $\mathcal{L}$-valeur de plusieurs noms. La possibilité de définir des "equivalences" en Fortran, les clauses "redefines" en Cobol et "share" en B.P.L. sont précisément d'autres moyens

de déclarer explicitement que l'on veut faire correspondre une même location à plusieurs noms (et plusieurs objets de nature éventuellement différente). Donc $\mathcal{L}$ serait plutôt le graphe Id x L qu'une fonction appliquant Id sur L de façon bijective.

2.3. Locations

Telles qu'elles ont été définies au paragraphe A.1., les différentes locations sont indépendantes. C'est pour cette raison que la fonction "Update" a été définie de la sorte :

Si $\sigma' = \text{Update}\ (\alpha, \beta)(\sigma)$

alors $\text{Contents}\ (\alpha)(\sigma') = \beta$

et $\text{Contents}\ (\alpha')(\sigma') = \text{Contents}\ (\alpha')(\sigma)$ si $\alpha' \neq \alpha$

Supposons maintenant que l'on déclare un tableau :

real array A [1 : n] ;

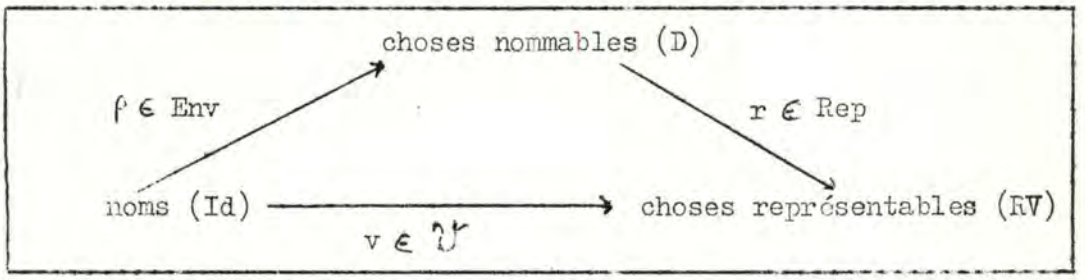
dans ce cas, A est le nom d'un objet L et A [i] d'un objet L, ce seront, d'après Strachey, des locations différentes, indépendantes. Or si l'on modifie A [i] par assignation, on aura donc $\text{Contents}\ (\alpha)(\sigma') = \beta$ si α est la $\mathcal{L}$-valeur de A [i], mais il est faux de dire que $\text{Contents}\ (\alpha')(\sigma') = \text{Contents}\ (\alpha')(\sigma)\ \forall\ \alpha' \neq \alpha$, car si α' est la $\mathcal{L}$-valeur de A, il est évident que modifier la $\mathcal{R}$-valeur d'un élément d'un tableau change la $\mathcal{R}$-valeur du tableau lui-même.

Il en est de même pour les structures, les listes..., bref pour tous les objets "composés". En fait, ce qui se passe, c'est qu'il y a partage implicite d'une même zone de mémoire mais que, contrairement au cas exposé au paragraphe B.2.2. b, où les différents noms avaient des "ranges" différents et où modifier une $\mathcal{R}$-valeur associée ne modifiait pas nécessairement les autres, ici, le "range" est le même et le recouvrement effectif. Les locations ne sont donc pas indépendantes.

Rappelons que si l'on se place au niveau du langage proprement dit, l'implémentation ne nous intéresse pas. Nous pouvons donc laisser de côté le bas du diagramme.

Nous obtenons ainsi le diagramme II.B.3.

II.B.3



où v associe à un nom la chose nommée, ou la représentation en termes de choses représentables de la (ou des) chose(s) nommable(s) associée(s). Donc, r, v et ρ sont des graphes.

Exemples :

v(-15.12) = ⟦-15.12⟧ où ⟦ ⟧ indique qu'il s'agit de l'objet réel désigné par le contenu des crochets.

Si l'on a déclaré int a = 3; int b = a; en Algol 68

et procedure p..... en Algol 60,

on a v(a) = v(b) = ⟦3⟧

v(p) = suite d'instructions correspondant à la procédure.

2.4. Dynamique du diagramme II.B.2.

Vu les définitions de l'assignation et des différents éléments intervenant dans ce diagramme, le seul élément variable au cours d'un même programme en est σ.

Or, en dehors de tout programme, Id ne contient que les noms standards définis par le langage et rien d'autre, et cela quel que soit le langage. Pour un programme donné, Id croît au fur et à mesure des déclarations et décroît à la fin des "ranges" des différents noms pour se retrouver à l'état minimun en fin de programme. Id est donc dynamique et il en est par conséquent de même pour toutes les "flèches" qui en partent et pour $L = \{\mathcal{L}\text{-valeurs de Id}\}$.

Dès qu'un langage est un tant soit peu extensible, il est évident que D et RV ne peuvent être fixes mais peuvent gonfler suivant les désirs du programmeur, dans les limites imposées par le langage.

Puisque Id, D, L, RV et σ sont variables, il en est de même pour ρ, $\mathcal{L}$, $\mathcal{R}$ et r.

Aux corrections émises en 2.1 et 2.2, il faut encore ajouter que $\mathcal{R}'$ est un graphe lui aussi car, si à une location correspond une seule $\mathcal{R}$-valeur à un instant donné, à un nom peuvent correspondre plusieurs locations, donc plusieurs $\mathcal{R}$-valeurs et réciproquement.

Nous pouvons donc conclure que le diagramme II.B.2. est dynamique, c'est-à-dire variable dans le temps, et que les flèches y représentent des graphes, sauf σ qui est une fonction.

C. Essai de généralisation

Nous allons maintenant essayer d'appliquer la méthode de classification donnée par Strachey à deux langages différents d'Algol 60. Tout d'abord, nous nous intéresserons à BCPL, un langage construit à partir des notions de $\mathcal{L}$- et $\mathcal{R}$-valeurs et donc du mécanisme de l'assignation tel que l'a analysé Strachey. Ensuite, nous étudierons Pascal, un langage de type Algol, mais permettant une grande variété de domaines dérivés.

1. B.C.P.L. (Basic Combined Programming Language)

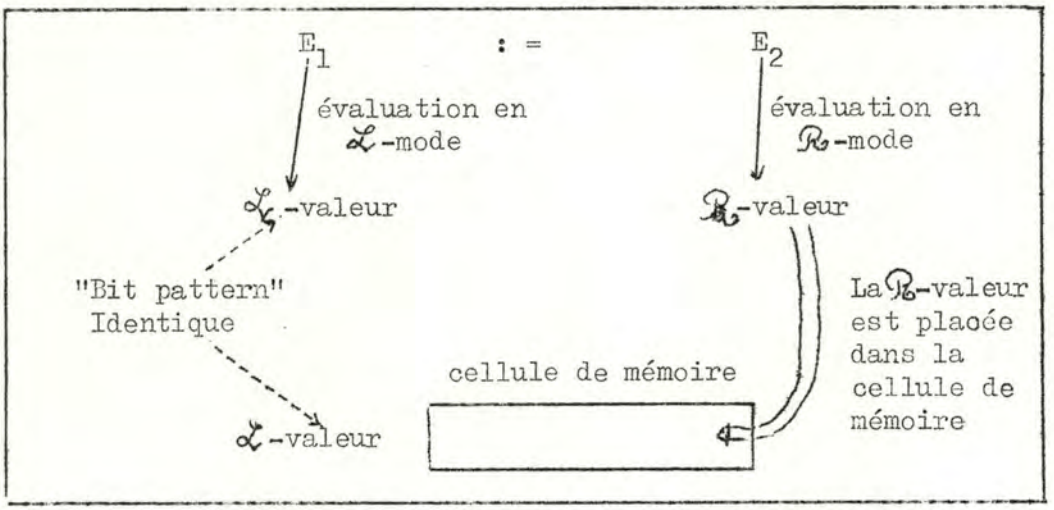
B.C.P.L. ou Basic C.P.L. est un langage simple, particulièrement adapté à traiter des problèmes non numériques assez complexes pour lesquels il est important de travailler indépendamment de la machine. Son application première est la compilation. Il a été conçu par M. Richards vers 1969 et remis à jour en 1971. Il se base sur un langage plus développé, C.P.L., dont Strachey est un des auteurs (1963).

1.1. Les grandes lignes du langage

La mémoire est divisée de façon idéale en cellules successives, étiquetées par des entiers. Ces entiers sont appelés $\mathcal{L}$-valeurs des cellules. Le seul mode de base, implicite, est celui des $\mathcal{R}$-valeurs (bit pattern), qui sont représentées sous forme d'entiers dans le langage. Donc $\mathcal{L}$- et $\mathcal{R}$-valeurs ont même représentation en B.P.C.L. L'assignation est de la forme :

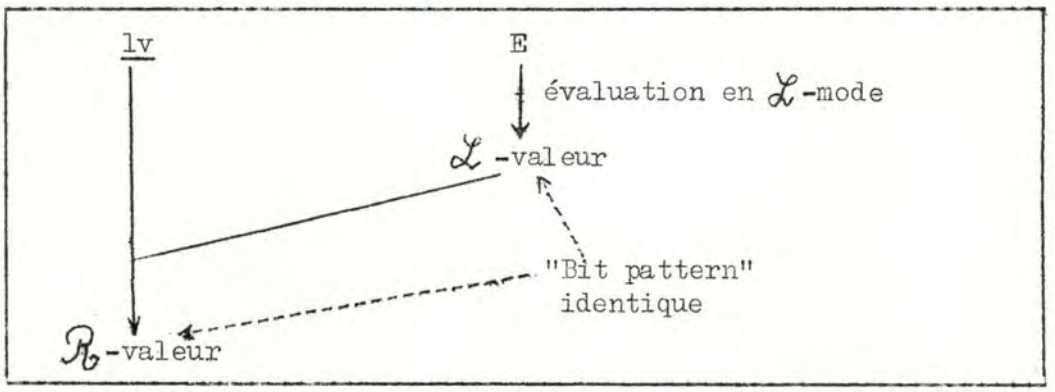
$E_1 := E_2$ où E_1 et E_2 sont des expressions.

L'exécution se fait comme suit :

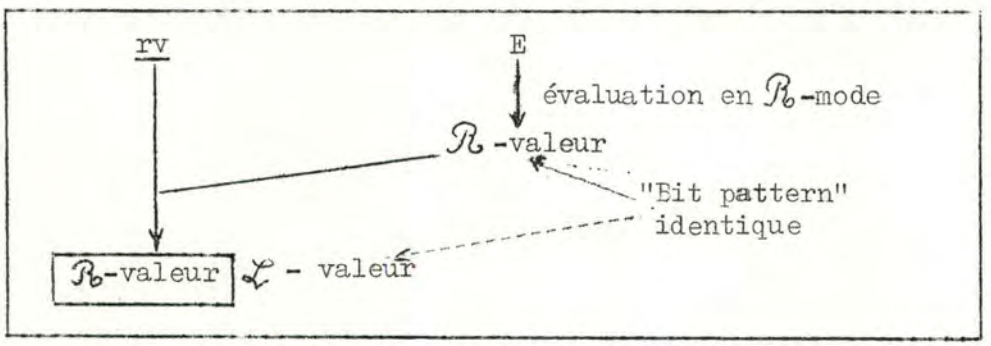


II.C.1.

Il existe deux <u>opérateurs</u> importants, <u>lv</u> et <u>rv</u>, qui permettent d'obtenir de façon explicite les $\mathcal{L}$-valeurs et $\mathcal{R}$-valeurs d'une expression et de s'en servir indifféremment comme $\mathcal{R}$-valeur et $\mathcal{L}$-valeur.
Les diagrammes II.C.2. et II.C.3. en donnent le schéma d'évaluation. On voit que l'utilisation de <u>lv</u> en partie droite d'une assignation permet d'assigner à une variable la $\mathcal{L}$-valeur d'une expression.

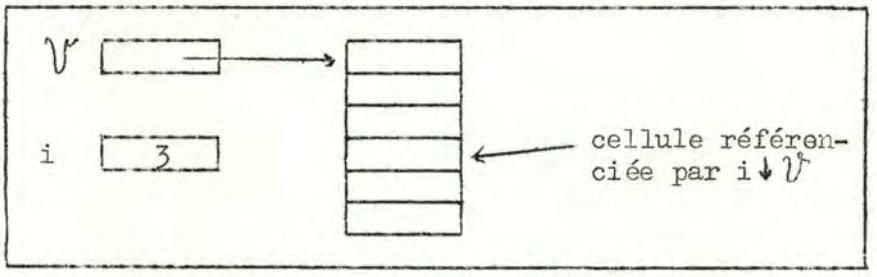


II.C.2.



II.C.3.

Quant à rv, son utilisation en partie gauche donne la cellule de mémoire dont la $\mathcal{L}$-valeur est identique à la $\mathcal{R}$-valeur de l'expression, tandis que son utilisation en partie droite donne le contenu de cette cellule.
Donc, une instruction rv p : = 1 mettra à 1 le contenu de la cellule dont l'adresse est la $\mathcal{R}$-valeur de p.
Cet opérateur, outre son utilité en compilation, permet la définition de "data structures" et de vecteurs.
Si on définit $\mathcal{V}\downarrow i$ = rv ($\mathcal{V}$ + i), i entier, on voit que $\mathcal{V}\downarrow i$ donne la $i^{ème}$ cellule d'un vecteur dont la cellule numéro 0 est indiquée par la $\mathcal{R}$-valeur de $\mathcal{V}$.
Mais on a : $\mathcal{V}\downarrow i$ = rv ($\mathcal{V}$ + i)
= rv (i + $\mathcal{V}$) car + est commutatif
= i $\downarrow \mathcal{V}$
Donc, $\mathcal{V}\downarrow i$ et i $\downarrow \mathcal{V}$ sont sémantiquement équivalents.
Toutefois, on considère V $\downarrow$ i comme la $i^{ème}$ composante d'un vecteur et i $\downarrow \mathcal{V}$ comme un sélecteur appliqué à une structure.



Les opérateurs +, -, * et / sur les $\mathcal{R}$-valeurs sont définis de manière à se modeler sur les opérateurs entiers.
Les autres opérateurs sont =, ≠, ls (<), gr (>), le (≤), ge (≥), not, ∧, ∨, ≡, ≢, lshift, rshift, + et - monadiques et rem qui donne le reste de la division de l'opérande de gauche par celui de droite.
Les expressions conditionnelles s'écrivent

$E_1 \longrightarrow E_2, E_3$ c'est l'équivalent BCPL de l'expression Algol 60
if E_1 then E_2, else E_3.

Pour plus de détails concernant les instructions existantes, voir la syntaxe de B.C.P.L. en Annexe II et "The B.C.P.L. Reference Manual" de M. Richards.

1.2. Classification suivant la technique de Strachey

- Domaines de Base :

Nous avons LV les locations ou $\mathcal{L}$-valeurs, J les "jump points", S les états de la mémoire et comme seul domaine de "données" proprement dites : RV ou $\mathcal{R}$-valeurs.

Remarquons que RV contient les booléens, true étant la $\mathcal{R}$-valeur dont tous les bits sont à 1 et false son complément, c'est-à-dire tous les bits à 0. Les caractères, lettres et symboles spéciaux ont un "code entier" (qui dépend de l'implémentation) et donc appartiennent à RV de ce fait.

- Domaines Dérivés :

E = D + V valeurs possibles des expressions.

Les vecteurs et les tables sont définis et implémentés sous forme d'une cellule contenant un pointeur vers une suite de cellules de mémoire.
On a donc

$$Pt = [LV \times S \longrightarrow (LV^*) \times S]$$

Les chaînes de caractères : La $\mathcal{R}$-valeur d'un string est un pointeur vers un ensemble de cellules de mémoire consécutives contenant la longueur et les caractères de la chaîne sous une quelconque forme compactée.

Donc $Q = [LV \times S \longrightarrow (LV^*) \times S]$

Les routines et les fonctions : l'appel d'une fonction ou routine est de la forme $E_o(E_1,\ldots,E_N)$ où les E_i désignent des expressions.
Le diagramme II.C.4 nous indique le processus d'appel d'une fonction ou routine. On constate que les paramètres sont appelés par valeur. Il est toutefois possible de simuler l'appel par référence par l'emploi des opérateurs lv et rv.
Ce qui donne

$$F = \underbrace{[E^* \longrightarrow [S \longrightarrow RV \times S]]}_{\text{fonctions}} + \underbrace{[E^* \longrightarrow [S \longrightarrow S]]}_{\text{routines}}$$

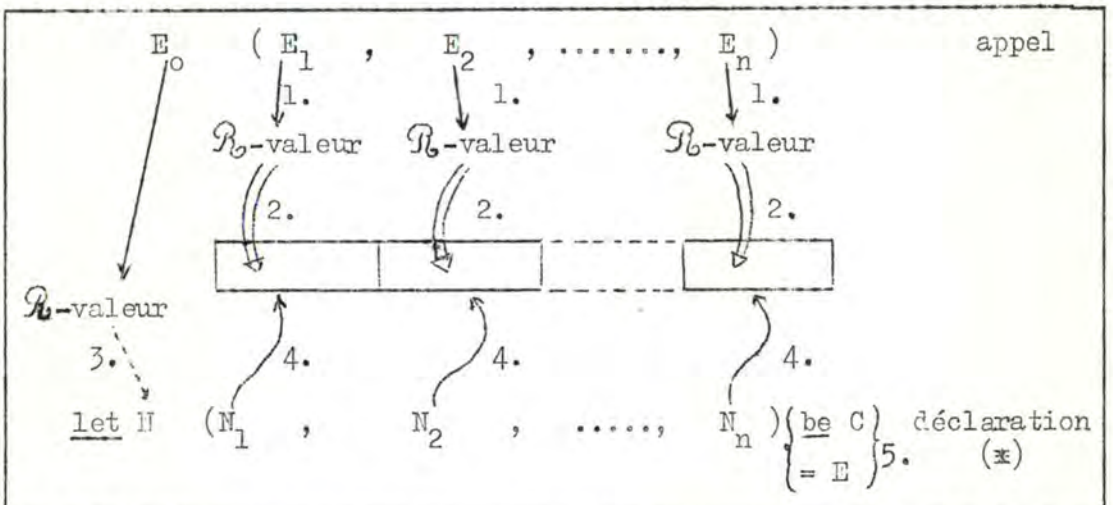
- Domaines caractéristiques :

Les objets stockables : V = RV + LV

V contient LV à cause de l'opérateur lv et de la concordance des représen-

tations en "bit pattern" des $\mathcal{L}$-valeurs et $\mathcal{R}$-valeurs.

II.C.4.



1. Evaluer les arguments en $\mathcal{R}$-mode.
2. Placer les $\mathcal{R}$-valeurs dans n nouvelles cellules de mémoire consécutives.
3. Trouver la fonction ou la routine correspondant à la $\mathcal{R}$-valeur de E_o.
4. Associer les cellules de mémoire de la gauche vers la droite aux paramètres formels.
5. Evaluer ou exécuter le corps de la fonction ou de la routine dans l'environnement de la définition, étendu par les nouvelles variables.

6a. Pour un appel de routine, revenir au point juste après l'appel.

6b. Pour l'application d'une fonction, produire comme résultat la $\mathcal{R}$-valeur du corps de la fonction.

(*) Lors de la déclaration, une variable est créée qui aura pour nom N et sera initialisée à C ou E suivant le cas, où C désigne une commande et E une expression.

Les objets nommables : D

D = RV constantes ("manifest declarations") et "case labels"
+ LV variables et paramètres (appelés par valeur!)
+ J "labels"
+ F fonctions et routines
+ Pt vecteurs et tables
+ Q "strings"

- Classification :

Si l'on rassemble ce qui précède, on obtient :

Domaines de base :
$$\begin{cases} LV \\ RV \\ J \\ S \end{cases}$$

Domaines Dérivés :
$$\begin{cases} V = LV + RV \\ E = D + V \\ D = RV + LV + J + F + Pt + Q \\ F = \left[E* \longrightarrow \left[S \longrightarrow RV \times S\right]\right] + \left[E* \longrightarrow \left[S \longrightarrow S\right]\right] \\ Pt = \left[LV \times S \longrightarrow (LV*) \times S\right] \\ Q = \left[LV \times S \longrightarrow (LV*) \times S\right] \end{cases}$$

- Conclusion :

La différence entre Q et Pt n'apparaît pas très clairement, à première vue, et il nous semble qu'une façon de les distinguer dans une schématisation de cette sorte serait de donner les fonctions primitives ou opérateurs qui peuvent agir sur ces domaines. En effet, si un élément quelconque d'un vecteur ou d'une table peut être mis à jour, il n'en est pas de même pour un "string" où les parties ne sont pas isolables puisque compactées en mémoire et où la modification d'une partie n'a pas de sens dans un tel langage. (Elle en aurait dans un langage de manipulation de strings ou dans un macro-générateur).

2. PASCAL

Décrit par N. Wirth en 1970, le langage de programmation PASCAL a été développé sur base d'Algol 60. Comparé à Algol 60, son domaine d'applications est considérablement plus étendu vu ses facilités de définition de modesvariés. Il a été conçu à la fois pour l'écriture de programmes complexes et comme base pour l'enseignement de la programmation.

2.1. Types de Données et Expressions

Il existe en Pascal cinq types de données définis et donc standard. Il s'agit de integer N, real R, boolean T = {true, false}, char C dénotés par des caractères entourés de quotes et alfa A = C^n séquences de n caractères, où n est un paramètre dépendant de l'implémentation.

On peut en outre définir un certain nombre de modes que l'on peut nommer et sur lesquels on peut définir des variables. Ils sont de différents genres : scalaire, sous-ensemble, tableau, structure, puissance, fichier, classe, pointeur.

- "Scalar type" : formés par énumération des éléments, dans l'ordre. Des fonctions standard succ et pred permettent d'obtenir le successeur et le prédécesseur (respectivement) d'un élément en suivant l'ordre d'énumération.
  Exemple : Color = (red, yellow, green, blue)

- "Subrange type" : sous-ensemble d'un type scalaire déjà défini ou standard.
  Exemple : weekday = monday .. friday
  avec week = (monday, tuesday, wednesday, thursday, friday, saturday, sunday)
  ou encore 20 .. 80.
  En effet, si l'on déclare une variable comme appartenant à un mode dont on donne la définition et pas le nom, ce mode n'en existe pas moins. Définir un mode suffit à assurer son existence, le fait de pouvoir le nommer est une extension supplémentaire. Dans ce cas-ci, 20.. 80 définit sans équivoque un mode du genre subrange.

- "Array type" : en Pascal, les tableaux sont, comme en Algol 60, composés d'un nombre fixe d'éléments de même type. Mais, contrairement à Algol 60, les index ne sont pas nécessairement de type entier, c'est pourquoi on précise le type des index dans la déclaration.
  Exemples : - déclaration d'un mode :
  card = array [1..80] of char.
  le type de l'index est un subrange de integer et le type des éléments est char.

- déclaration d'une variable dont le mode n'a pas été nommé (ce qui n'est pas nécessaire, comme nous l'avons fait remarquer un peu plus haut) :

  p : array [Boolean] of Color

  Le type des éléments est Color et celui de l'index est Boolean. Donc, si à un instant donné p est (red, green), alors

  p [true] = red

  et p [false] = green

- "record type": une structure se compose d'un nombre fixe de composants, appartenant à des types éventuellement (et généralement) différents. Dans la définition d'un mode de genre "record", on déclare ou définit le type de chaque composant ("field"). De plus, chaque composant est nommé; ce nom, dont le "range" est la définition du mode, servira de "field selector". Une particularité assez intéressante des records est la possibilité d'avoir des "fields" variables. Un "field" variable sera annoncé par un mot spécial : case, on déclare ensuite un "tag field" qui n'est autre qu'un identificateur dont on précise le mode; ensuite vient l'énumération d'un certain nombre de "fields" ou groupes de "fields", chacun précédé d'un identificateur ("case label") qui est le nom d'un objet particulier du type du "tag field".
  Ainsi, suivant la valeur de la variable "tag field", on choisira le "field" étiqueté par cette valeur.
  Dans l'exemple suivant, si s:= male, on choisira le premier groupe de "fields", sinon (s:= female) le second.
  Exemple : soit la définition suivante :

```
Sex = (male, female)
on a alors
Person = record name, firstname: alfa;
                age : integer;
                married : boolean;
                case s : Sex of
                    male: (enlisted, bold: boolean);
                    female: (pregnant: boolean)
           end
```

- "Powerset type" : ce type définit un domaine de valeurs comme l'ensemble des parties d'un autre type scalaire ou standard, appelé type de base. Les opérateurs qui s'y appliquent sont
  ∨union, ∧ intersection, - différence d'ensemble et in appartenance.
  Exemple : h : powerset Color

- "file type : ce type spécifie une séquence de composants, tous du même type. La longueur de cette séquence n'est pas fixée dans la définition. A chaque fichier, ou variable du type file, est associé un pointeur qui dénote un élément spécifique. Quand le fichier est en état input, output ou neutre (état initial), on peut modifier la position du pointeur par des procédures standard.
  Exemple : Charfile = file of char.

- "class type" : il s'agit, ici aussi, d'une séquence d'éléments de même type, dont le nombre est variable. Mais dans ce cas-ci, le nombre est nul lors de la déclaration de la variable. Les composants sont créés, c'est-à-dire de la mémoire leur est allouée, en cours d'exécution par une procédure standard "alloc". La déclaration de type spécifie cependant le nombre maximum d'éléments.
  Exemple : family : class 100 of Person

- "pointer type" : un mode "pointeur" est associé à toute variable de type "class". Les éléments en sont les pointeurs possibles vers les composants de la variable de type "class" et le pointeur nil qui ne désigne aucun composant. Un type pointeur est dit "lié" à sa variable de type "class".
  Exemple : p_1 : ↑ family.

- Expressions : la différence essentielle par rapport aux expressions d'Algol 60 est la possibilité de construire des "sets", c'est-à-dire des listes d'expressions de même type.
  Exemple : [red, c, green].
  Il est ainsi possible d'assigner des valeurs à des variables de type "powerset".
  De plus, il n'existe pas d'expressions de désignation. La syntaxe complète de Pascal se trouve en Annexe III.

2.2. Classification de Pascal

- Domaines de base :

S états de la mémoire
L locations
J "jump points"
T booléens
N entiers
R réels
C caractères
C^n "alfa" (n; paramètre de l'implémentation)
Id identificateurs

Ce dernier domaine, Id, est de base car il sert de matériau de construction pour les types scalaires.

- Domaines dérivés :

Les variables de type Powerset : (ensemble des parties d'un type scalaire, standard ou subrange).

Pw = $[S \longrightarrow (L\text{æ}) \times S]$ en effet, si h : Powerset Color avec Color = (red, green, blue), alors on peut écrire h: = [red, blue] par exemple.
h pourra avoir comme $\mathcal{R}$-valeur une partie quelconque de $\mathcal{P}$(color) donc : [red, green, blue] , [red, blue] , [green, blue] , [red, green] , [red] , [green] , [blue] .

Le nombre de locations nommées par h varie donc suivant S .

Les tableaux : On ne peut plus ici se contenter de $L\text{æ}^*$. En effet, les éléments d'un tableau doivent tous être de même type mais ce type peut être "class", "file", "pointer", "powerset", "record" aussi bien que "real", "integer",... Or, comme nous allons le voir dans ce qui suit, les variables de ces différents types ne sont pas toujours des locations. Nous avons donc:
A = Læ + CFLæ + Reæ + Ptæ + Pwæ + Aæ
c'est-à-dire : tableaux de locations (1 dimension, les suivantes sont données dans Aæ), de variables de type "class" et "file" (CFL), "record" (Re), "pointer" (Pt), "powerset" (Pw) et "array" (A).

Les variables de type "class" et "file" :

$CFL = [S \longrightarrow A \times S]$ la longueur étant variable, elle est déterminée dynamiquement en fonction de S et puisque les éléments sont de type quelconque mais tous du même, nous avons besoin de A pour désigner une telle liste d'éléments.

Les variables de type "pointer" :

$Pt = [L \times S \longrightarrow A \times S]$ un pointeur pointe vers une variable de type class.

Exemple : family : class 100 of Person

p : ↑ family p pointe vers family

et p ↑ family pointe vers un élément de cette classe.

Les variables de type "record" :

$Re = (D' + W_1)^*$ où $D' = L + A + Re + CFL + Pt + Pv$

c'est-à-dire l'ensemble des variables quel que soit leur type et

$W_1 = [D' \times S \longrightarrow [(Sc \setminus R) \longrightarrow D' \times S]]$ le domaine des "variant part" dans un "record".

$Sc = R + N + T + C + C^n + Id$ c'est-à-dire l'ensemble des constantes.

En effet, ces parties variables se présentent sous la forme suivante :

```
case S : <type> of
<case label> : (... : .....);
      ⋮
<case label> : (... : .....);
```

S est le "tag field" qui est d'un type quelconque, donc $\in D'$. On regardera quelle est la valeur de S et on la comparera aux différents "case label" pour obtenir le "field" désiré.

On a donc : $[D' \times S \longrightarrow [(Sc \setminus R) \longrightarrow D' \times S]]$

- D' : tag field
- $(Sc \setminus R)$: case label (constante sans signe, R désigne les réels sans signe)
- $D' \times S$: field choisi

Les "calls by name" :

$W = [S \longrightarrow (D' + F) \times S]$ les paramètres soit ont un type, donc $\in D'$,

soit sont des procédures ou des fonctions, donc $\in$ F (voir ci-dessous).

Les procédures et fonctions : le résultat d'une fonction ne peut être que de type standard, scalaire, subrange ou pointeur.

On a donc :

$$F = \underbrace{\left[(D' + F)^{*} \longrightarrow [S \longrightarrow S]\right]}_{\text{procédures}} + \underbrace{\left[(D' + F)^{*} \longrightarrow [S \longrightarrow (Sc + Pt) \times S]\right]}_{\text{fonctions}}$$

On obtient ainsi la classification suivante pour les domaines dérivés :

- le domaine des choses nommables :

$D = T + N + R + C + C^{n} + Id$	constantes
$+ L$	variables de type scalaire, standard et subrange
$+ A$	tableaux
$+ Re$	variables de type record
$+ CFL$	variables de type class et file
$+ Pt$	variables de type pointeur
$+ Pw$	variables de type powerset
$+ J$	jump points
$+ F$	fonctions et procédures
$+ W$	calls by name
$+ W_1$	variant part dans les records.

où A, Re, CFL, Pt, Pw, F, W et W_1 ont été définis plus haut.

- le domaine des choses stockables :

$V = Sc + Pt + E^{*}$

E^{*} : sets

- le domaine des valeurs possibles des expressions :

$E = D' + V = D' + Sc + E^{*}$

Remarque : Nous n'avons pas réussi à formaliser l'existence d'un pointeur lié à un fichier ainsi que son état d'entrée, de sortie ou neutre. N'apparaissent pas dans cette classification non plus la présence d'un mode pour les index des tableaux et les différentes possibilités de nommer des modes. Nous avons simplement voulu nous conformer à la méthode dd Strachey et donc nous intéresser aux domaines et à leurs éléments et pas aux modes.

3. Conclusions

Ces deux essais de classification nous ont suggéré les conclusions suivantes : nous ne voyons pas comment il serait possible de classifier des langages plus extensibles qu'Algol 60 ou BCPL au moyen de l'outil de schématisation proposé par Strachey. Il nous semble difficile en outre d'incorporer les opérateurs dans son modèle, tel qu'il est défini. Il nous paraît cependant intéressant, dans le but que nous poursuivons, de les prendre en considération.

Nous allons maintenant reprendre en détail ces deux conclusions dans les paragraphes qui suivent.

3.1. Opérateurs et fonctions primitives

La structure induite dans un langage par les opérateurs nous semble justifier une étude plus détaillée de ceux-ci.

En effet, les opérateurs sont des objets qui font partie du langage au même titre que ceux auxquels Strachey s'est intéressé et que nous appellerons "opérandes" dans la suite de ce paragraphe. Leur rôle est cependant un peu différent : ils sont les outils de manipulation des opérandes. Les opérateurs arithmétiques, logiques et relationnels relient les opérandes pour former les expressions. Les déclarateurs (qui sont généralement les noms des modes ou des domaines) servent à définir de nouveaux identificateurs, modes ou objets.

Ce sont ensuite les compléments indispensables aux éléments des domaines pour former des modes. En effet, un mode est un ensemble d'objets appartenant à un même domaine, ayant des propriétés communes et d'opérateurs agissant sur les éléments de ce domaine.

Strachey s'intéresse plus, nous semble-t-il, aux domaines et à leurs éléments qu'aux modes. Il est donc vraisemblable que ce soit une des raisons pour lesquelles il n'a pas fait figurer cet aspect des langages dans sa classification. Il est assez intéressant de remarquer que, tout comme pour les opérandes, on peut regrouper les opérateurs en différents domaines de base et dérivés. Seuls les langages extensibles, toutefois, permettent de définir

de nouveaux opérateurs. Mais tous les langages possèdent un lot d'opérateurs standard définis et nommés, donc de base.

Une simple réflexion suffit pour se persuader que les opérateurs, qu'ils soient de base ou dérivés, ont une $\mathcal{P}$- et une $\mathcal{L}$-valeur, respectivement la routine d'exécution et son emplacement en machine.
Certains langages permettent de donner un autre nom à des opérateurs standard. On remarquera qu'un même opérateur peut donc éventuellement avoir plusieurs noms et que, de plus, un même nom peut désigner différents opérateurs. Exemple: + en Algol 60 est à la fois l'addition entière et réelle, le contexte seul permet de décider.

Nous pensons qu'il pourrait être important de faire figurer les opérateurs standard, groupés par domaines, dans l'énumération des domaines de base. Sinon, comment exprimer dans D que des opérateurs peuvent être nommés par le programmeur, si tel est le cas ? De plus, il serait dommage de ne pouvoir distinguer dans leur classification deux langages qui offriraient les mêmes possibilités du point de vue opérandes mais seraient totalement ou partiellement différents quant aux opérateurs.
Remarquons que, dans les langages extensibles, où l'on peut définir des opérateurs, ceux-ci peuvent devenir à leur tour des opérandes. Nous approfondirons ce point au chapitre quatre.

Parmi les opérateurs, il y a peut-être lieu de distinguer les fonctions primitives qui sont de deux sortes. Tout d'abord, les fonctions standard, définies et nommées, donc explicites. Exemple: sin, cos, abs,...
Ensuite, les fonctions implicites, définies mais non nommées et non nommables. Il s'agit généralement, dans ce second cas, des conversions de type (coercitions), de la sélection d'un élément d'un tableau, d'un "field" d'une structure,...

Revenons à BCPL, nous y trouvons $Q = [LV \times S \longrightarrow (LV\ast) \times S]$ et
$Pt = [LV \times S \longrightarrow (LV\ast) \times S]$.
Nous avons déjà signalé ce problème au paragraphe B.1.2. Il serait immédiatement résolu si la fonction primitive de sélection d'un élément d'un vecteur, qui en BCPL est l'opérateur $\downarrow$, était signalé dans la classification du langage. C'est pourquoi, vu l'intérêt de ce problème dans le cadre de

notre étude des objets nommables, nous avons essayé de classifier BCPL dans son entièreté. Nous avons dû, pour ce faire, étendre la méthode de Strachey aux opérateurs (voir Annexe IV). Nous n'avons malheureusement pas pu réaliser cet objectif pour Pascal. Il nous semble que Pascal est un peu trop complexe pour pouvoir être classifié complètement de cette façon. Il serait possible que la méthode proposée par Strachey ne soit pas parfaitement adaptée à l'objectif que nous poursuivons et plus particulièrement à la classification de langages extensibles complexes. Ceci rejoint la seconde conclusion que nous abordons maintenant.

3.2. Extensibilité

Les outils définis par Strachey permettent-ils de générer tout domaine dont nous pourrions avoir besoin ? Il s'agit, rappelons-le, des opérations ensemblistes +, x, * et $\longrightarrow$. Notre but est de pouvoir exprimer tout domaine dérivé en fonction des domaines de base, et principalement le domaine de tous les objets nommables, et ce pour n'importe quel langage, y compris des langages extensibles comme PL/1 ou Algol 68.
Par "tous les objets nommables", nous entendons les opérateurs, les "opérandes" et les modes.

En appliquant la méthode de Strachey à Pascal, nous ne sommes parvenus qu'à classifier les "opérandes". Il semble que l'emploi d'opérateurs ensemblistes tels que $\cup$ (union), $\cap$ (intersection), $\setminus$ (différence), $\mathcal{P}$ (ensemble des parties), $\complement$ (complément), $\in$ (appartenance) et $\{\ldots\}$ (construction d'un ensemble par énumération) pour générer des domaines pourrait donner une plus grande souplesse à une schématisation de ce genre. En effet, certains d'entre eux sont utilisés en Pascal ($\mathcal{P}$ et $\{\ldots\}$ pour ne citer que ceux-là) et plus les langages seront extensibles, plus on s'en servira; de plus, ils ne semblent pas tous facilement exprimables en termes de +, x, * et $\longrightarrow$. Nous nous sommes, en fait, explicitement servis de $\setminus$ pour classifier Pascal.

Pour des langages encore plus extensibles, on arrive très vite à schématiser différents domaines un peu complexes sous la même forme D'*. Il nous semble que ce n'est plus assez précis et à la limite permettrait

peut-être même de définir des domaines qui n'existeraient pas vraiment ou seraient trop larges dans le langage.

Ainsi que nous l'avons déjà soulevé à plusieurs reprises, en Pascal et dans plusieurs autres langages il est possible de définir et même de nommer des modes. D'autre part, ainsi que le montrent les tableaux T.II.A.1 et T.II.A.2, certains domaines de base possèdent un mode associé, défini et nommé. Ces modes peuvent, dans quelques langages, être renommés par le programmeur et sont donc aussi nommables. Nous souhaiterions pouvoir tenir compte de ces faits dans notre classification.

Une notation du genre $\mathbb{N}$, où $\mathbb{N}$ serait le mode entier, pourrait éventuellement résoudre le second problème. Ainsi en Pascal, où l'on peut déclarer a = 3 mais aussi entier = integer, D devrait contenir à la fois N et $\mathbb{N}$!

Par contre, pour les modes créés par le langage lui-même, la solution n'est pas aussi aisée et il nous a semblé fort difficile de schématiser une telle possibilité. Nous avons essayé de le faire pour Pascal (voir Annexe V) et nous avons eu besoin de notations supplémentaires.

En conclusion, il nous semble que la méthode de classification proposée par Strachey, même affinée, est difficilement applicable à l'étude des propriétés de nommabilité dans un langage. En effet, cette méthode étant basée uniquement sur l'étude du domaine de définition et du "range" de certaines fonctions à l'exclusion de toute autre propriété, toute caractéristique du langage qui ne découlerait pas de ces propriétés ne pourrait y être mise en évidence. Nous croyons avoir montré que, dans le cadre cependant restreint de la nommabilité, de telles caractéristiques existent et sont même d'autant plus nombreuses que le langage est plus complexe.

Chapitre Trois

LA CLASSIFICATION INDUITE PAR ALGOL 68

Parmi les langages extensibles, nous nous proposons d'étudier maintenant Algol 68.

Prévu comme un successeur d'Algol 60 dès 1963, Algol X devint Algol 68 quand, en décembre 1968, les dernières mises au point y furent apportées. C'est un langage général, qui regroupe la plupart des particularités rencontrées dans les autres langages et qui permet une assez grande souplesse et extensibilité.

Dans ce chapitre, nous ferons surtout référence au "Report on the Algorithmic Language ALGOL 68" (R.) et à l'"Informal introduction to Algol 69" (I.I.).

A. Les grandes lignes d'Algol 68

Dans le cadre de notre étude des langages sous l'angle de la nommabilité, nous porterons principalement notre attention sur les objets nommables et nommés d'Algol 68.

1. Assignation

Au point de vue sémantique, la notion d'opération, en Algol 68, est plus générale qu'à l'habitude : elle est analogue à celle de procédure à un ou deux paramètres. Une opération possède ou non un résultat (autrement dit, elle constitue soit le calcul d'une expression, soit l'exécution d'une instruction); dans les deux cas, elle exécute certaines actions. On les appelle "actions primitives" ou "propositions unitaires" (R. et II.).
C'est ainsi que l'assignation est une opération de confrontation qui admet deux opérandes, la source, à droite du ":=" (":=" est le nom de l'opération), et la destination, à gauche. Si la destination est de mode ref μ (voir 2.2),

l'évaluation de la source doit fournir une valeur de mode μ. Pour ce faire, on utilisera éventuellement des déréférentiations ou des coercitions (voir 2.3).

L'opération se déroule en trois étapes : (R. 8.3.1.2d.)

1) évaluer les opérandes en parallèle,
2) après déréférentiation ou coercition éventuelle, affecter la valeur de la source à la valeur de la destination (qui est un nom : voir B.2.1.),
3) l'assignation possède en elle-même une valeur qui est celle de sa destination.

On remarquera que l'assignation est une opération qui produit une valeur. Il est donc normal de considérer que, du point de vue sémantique, l'assignation se comporte en tous points comme un opérateur : possède deux opérandes, les évalue, effectue une certaine action sur eux et produit un résultat.
Il s'agit donc d'un point de vue assez différent de celui exprimé par Strachey (voir ch. IV).

2. Classification

Pour plus de concision, nous allons classifier directement les objets nommés et nommables en Algol 68, tout en détaillant certains points. Nous allons considérer deux groupes d'objets : les objets de base, opérandes et opérateurs, et les objets dérivés. Pour ce faire, nous restons toujours fidèle à la définition des domaines de base, donnée au chapitre II (A.2.).

2.1. Domaines de Base

Il s'agit des ensembles primitifs de valeurs à partir desquels on pourra construire d'autres domaines.

- Opérandes

Comme dans beaucoup de langages, et en particulier en Algol 60, parmi les domaines de base, on trouve les réels, les entiers et les booléens. De plus, les caractères et, fait particulier à Algol 68, les formats constituent eux aussi des domaines de base. Soulignons enfin qu'en Algol 68, tous ces domaines constituent des modes et que leurs éléments sont donc assignables. En particulier les formats peuvent être assignés à des variables,

donnés comme paramètres à des procédures et utilisés dans la composition de domaines dérivés.

Si l'on s'en tient rigoureusement à la définition, il n'y a pas d'autres domaines de base constitués d'objets du genre opérande à prendre en considération.

Certains domaines dérivés sont cependant définis dans le langage, ce sont les "bits", "bytes", "strings", "sema" et "complex".
Les valeurs ou objets de ces domaines sont définissables en termes de structures, tableaux et références à des domaines de base.

Bien que les objets de ces domaines soient définissables, ils sont aussi définis, et bien rares seront les programmeurs qui n'utiliseront pas la définition du langage pour ces domaines. C'est pourquoi, et c'est en fait une simple question de choix, nous les avons "assimilés" aux domaines de base.

- Opérateurs

Deux domaines distincts sont ici en présence. Il s'agit tout d'abord des actions primitives qui ont comme principales propriétés qu'elles ne sont pas nommables et qu'il n'existe pas de mode associé ni défini, ni définissable. Ensuite, nous avons les routines. Celles-ci sont nommables et il existe des modes correspondants. Un mode formé de toutes les routines n'existe cependant pas. En effet, tout mode formé de routines précisera toujours le nombre des paramètres, avec leurs modes respectifs et le mode du résultat s'il y en a un (sinon il est de mode vide : "void").

On pourrait se demander pourquoi on a séparé les routines des autres domaines de base cités dans la partie "opérandes", alors que les routines ont un mode, et sont donc assignables, peuvent être paramètres de procédures et composantes de domaines dérivés. En fait, il n'existe pas d'opérateurs sur les routines et celles-ci ne peuvent donc pas être combinées entre elles pour en produire d'autres, comme il est possible de le faire pour les réels, entiers, etc... Il n'est donc pas possible en Algol 68 de considérer les routines comme des opérandes.

Les opérateurs proprement dits sont donc des routines et de ce fait sont tous définis. Un certain nombre d'entre eux sont déjà nommés dans le

langage. Ils sont cependant aussi nommables. Lorsque le programmeur nommera un autre opérateur en déclarant par exemple :

op ◇ = (int a,b) int: if a ≤ b then b else a fi;

il ne définit pas un nouvel opérateur, puisque la routine est déjà définie, mais il la nomme. L'extension du langage réside donc non pas tant dans le fait que les routines sont nommables, mais bien plus dans l'existence de modes correspondant aux différents groupes de routines. (Nous avons appelé "groupe de routines" un sous-domaine des routines rassemblant celles qui concordent quant au nombre et aux modes des paramètres ainsi qu'au mode du résultat).

Nous allons maintenant détailler quelque peu les opérateurs nommés par le langage. Rappelons que ce sont donc des objets appartenant à un mode "routine".

- opérateurs arithmétiques : sur les entiers, réels, booléens et complexes.
- opérateurs logiques : sur les booléens et les bits.
- opérateurs relationnels ou de comparaison : certains sur les entiers, réels, booléens, caractères, bytes et strings, d'autres en plus sur les bits et quelques-uns sur les booléens et bits uniquement. Tous sont à résultat booléen.
- opérateurs combinés avec assignation (+:=, x:=, ...)
- conversions explicites (bin, round, entier, repr, btb, ctb, ...)
- opérateurs spéciaux associés à un seul mode :
  - sur les strings
  - sur les complexes (re, im, conj, ...)
  - sur les sémaphores (up, down,/)
  - sur les bits et bytes (elem).

Tous ces opérateurs sont décrits dans le Rapport : R.10.2.2 à 10.2.11 et R.10.4.

On remarquera que quelques-uns de ces opérateurs sont des routines faisant appel, sous forme de commentaire, à des opérations définies ailleurs. Il s'agit de "smaller than", "minus", "times", "divided by" et aussi d'équivalences entre entiers et réels et entre caractères et entiers positifs (R.2.2.3.1. c, d et f). Bien que ces routines ne soient pas dénotées explicitement

dans R, elles sont définies et nommées.
(< en R 10.2.3.a, - en R.10.2.3.g etc..)

- à ces opérateurs, il faut ajouter ce que l'on appelle généralement les fonctions standard. Ces fonctions sont des routines. La différence entre celles-ci et les opérateurs est une question de notation : le nom de l'opérateur se place entre les opérandes, lors de l'appel, et celui des fonctions devant les opérandes. Ces fonctions sont celles que l'on trouve traditionnellement dans tous les langages à applications numériques : cos, sin, tan, ln, exp, sqrt,...
  La remarque faite pour "minus"... juste un peu plus haut est aussi valable pour les fonctions standard (R.10.3).

Passons maintenant aux actions primitives.
Parmi celles-ci, se trouvent les opérations de confrontation, les coercitions et le test sur la valeur d'une condition booléenne.

- les opérations de confrontations, ce sont les seules actions primitives nommées.
  Elles comprennent l'assignation ":=" (R.8.3.1) dont nous avons parlé en A, les relations de conformité "::" et "::=" (R.8.3.2) et les relations d'identité ":=:" et ":≠:" (R.8.3.3).

- les coercitions et conversions implicites : les premières sont au nombre de huit : "dereferencing", "deproceduring", "proceduring", "uniting", "widening", "rowing", "hipping" et "voiding" (R.8.2).
  Etant donné que la plupart d'entre elles s'appliquent aux différents domaines dérivés, nous préférons expliquer, pour chaque domaine dérivé, les coercitions et conversions qui s'y rapportent. Ce sera fait au paragraphe 2.2.
  Restent "widening", "voiding" et "hipping" :
  * "widening" qui "élargit" un mode en transformant int en real
    long int en long real
    real en compl
    etc...

* "voiding" qui supprime le mode d'une expression. Le résultat de cette opération est que la valeur produite par l'expression sera totalement négligée.
* "hipping" qui donne un mode à une expression qui n'en a pas. exemple : goto 1 devient de mode proc. (proc est le mode des routines sans paramètres ni résultat)

- quant au test sur la valeur d'une expression booléenne, on le retrouve dans les conditionnelles (expressions et instructions).

Deux remarques avant de terminer ce paragraphe :

- Tout comme en Algol 60 et dans beaucoup d'autres langages, en Algol 68, un même nom peut désigner plusieurs opérateurs, donc plusieurs routines. Le contexte, c'est-à-dire généralement le mode des opérandes, permet de décider quel opérateur est à appliquer. Cela se traduit, en Algol 68, par le fait que toute routine a un mode qui diffère suivant les types et le nombre des opérandes et le type du résultat.

- Le rapport définit la priorité de tous les opérateurs nommés. Il s'agit d'un nombre entier compris entre un et neuf (R.10.20). Il laisse cependant au programmeur le soin de choisir la priorité des opérateurs qu'il nommera ainsi que la possibilité de modifier, pour l'étendue d'un bloc, la priorité d'un opérateur nommé par le langage.

2.2. Domaines dérivés

Remarquons tout d'abord qu'Algol 68 permet de définir des "long values" pour doubler, tripler, ... la longueur de la représentation mémoire d'un objet, afin d'augmenter la précision des calculs. Mais, vu que c'est principalement une question d'implémentation, nous n'en tiendrons pas compte dans notre classification.

Algol 68 fournit différents moyens de définir de nouveaux domaines et modes : constitution de tableaux, produits cartésiens d'ensembles (structures), références à des modes de base (opérandes et routines) et union de modes.

- Valeurs multiples : ces objets sont formés par regroupement de zéro, une ou plusieurs valeurs, toutes de même type. Chacune de ces valeurs est un élément de la valeur multiple (nous dirons aussi "multiple" pour "valeur multiple").
Chaque élément est sélectionné par un index (entier).
A chaque multiple est associé un descripteur qui indique comment les éléments en sont indexés et quels sont les degrés de liberté que l'on a pour les bornes. Ces bornes peuvent être fixes ou variables, ce qui sera indiqué par l'absence ou la présence du symbole flex à l'endroit voulu dans le descripteur.

Exemple : [h : k flex, m : n] real

On construit ainsi des tableaux de dimension quelconque dont il est possible d'extraire aussi bien les éléments que des parties ou "tranches".
Voici les coercitions et conversions agissant sur les multiples :

Construction par "row display".

Exemple : $(E_1, \ldots, E_n)$

Si les E_i sont de mode μ, alors $(E_1, \ldots, E_n)$ sera, par coercition, de mode $\in \exists \mu$. Donc le mécanisme de "row display" ajoute une dimension.
Sélection par "rowing". Ce mécanisme permet de sélectionner des parties ou "tranches" de valeurs multiples et de les considérer comme étant d'un autre mode du genre valeurs multiples, mais avec éventuellement une ou plusieurs dimensions de moins, ou des bornes réduites.

Exemple : t [,4] désigne la quatrième colonne de t.
t [2: , 1:3] désigne la partie du tableau t formée des t [i,j] tels que $2 \leq i$ et $1 \leq j \leq 3$.

- Valeurs structurées : ces objets sont formés par collection d'un certain nombre de valeurs, de modes qui peuvent être différents, appelées "fields" de la structure, chacun d'entre eux pouvant être sélectionné par un sélecteur de "field".
Les structures (ou valeurs structurées) recouvrent les objets connus dans d'autres langages sous les noms de "records", "lists", "trees", "queues", "chains", etc...

Les coercitions et conversions sont les suivantes :

Construction par "structure display".

$(E_1, \ldots, E_n)$, E_i de mode μ_i.

Exemple : struct (int a, real b) c = (n+10, 3.14)

Sélection par "field selection".

struct $(\mu_1 s_1, \mu_2 s_2, \ldots, \mu_n s_n)$

où μ_i sont des modes et s_i des identificateurs.

Si of μ_i sera de mode μ_i par coercition.

- Valeurs référencées : "names" dans R. et II. (Un autre mot tel que "référence" nous a semblé meilleur).

Si μ désigne un mode, alors ref μ désignera le mode des références repérant des valeurs de mode μ.

En fait, puisque ref μ sera lui-même un mode, les références seront donc aussi des valeurs et il est possible de définir d'autres objets qui repèreront des références d'un certain mode.

On peut ainsi avoir ref ref real par exemple.

En Algol 60, la déclaration "real x" a un double rôle : réserver la place mémoire (location) susceptible de contenir un réel et représenter l'identificateur x par l'adresse de cette location.

Par contre, en Algol 68, les deux effets sont séparés. Une location est la représentation hardware d'une référence. La valeur repérée par la référence est représentée par le contenu de cette location.

En quelque sorte, les identificateurs et les références seraient respectivement les accès externes et internes aux valeurs.

Il faut bien voir que le premier niveau de référence correspond à ce qui, dans les autres langages, est appelé "variable". Alors que les niveaux supérieurs correspondent à la notion de "pointeur".

Syntactiquement donc, en Algol 68, une "variable" est une référence à un mode. Sémantiquement, l'objet qui, réellement, est variable, est la valeur repérée par la référence.

On a donc la correspondance suivante :

unmode ⟷ chose, objet ou valeur de mode unmode

ref unmode ⟷ référence à une valeur de mode unmode

ref ref unmode ⟷ pointeur vers une valeur de mode unmode, c'est-à-dire référence à une référence à une valeur de mode unmode.

Ces valeurs peuvent être, entre autres, utilisées pour faire du chaînage et créer ainsi des listes de longueur variable.

Exemple : mode box = struct (amode value, ref box next); (II 1.4.3)

Il est donc possible, en particulier, de définir des références à des routines. On obtient ainsi les procédures. Celles-ci ont donc un mode et sont assignables.

Exemple : soit mode aproc = proc (real, ref bool) int une déclaration de mode,
proc a = ((real x, ref bool b) int : if b then sin (x) else cos (x) fi; une déclaration d'une constante a, donc d'une routine et
ref aproc p = loc aproc une déclaration de variable, donc d'une procédure.
On pourra alors assigner a à p :
p:= a et p désignera donc une location contenant a.

Les coercitions s'appliquant aux références comprennent celles pour les procédures :

création par "generator":loc pour une valeur locale et heap pour une valeur globale.

Dans le contexte d'une assignation, transformation d'un mode ref μ en mode μ par "dereferencing".

Exemple : ref real x,y; y:= 3.14;
x:= y

ce n'est pas la référence y mais le réel repéré par y qui sera assigné à x.

"proceduring"

Exemple : proc real p;
p:= x:= 3.14;

x:= 3.14 sera évalué et deviendra une routine qui assigne 3.14 à x et produit 3.14 comme résultat et c'est cette routine qui sera assignée à p.

"deproceduring"

Exemple : proc real p = 3.14 + x;
proc real q = p;
real y = p

à la troisième occurrence de p, il faut "déprocédurer" p.

- Union : il est parfois intéressant de ne pas définir exactement le mode d'un objet avant de l'employer. Or le mode ou le domaine sert entre autres à spécifier au compilateur la représentation interne de l'objet et la façon dont on peut le manipuler (opérateurs...). Aussi Algol 68 permet de définir des domaines comme étant l'union de plusieurs modes. Il s'agit d'une union disjonctive. Il est évident qu'au moment de l'exécution, un choix sera fait suivant un critère précisé dans le programme, afin qu'une valeur, à un instant donné, n'appartienne bien qu'à un seul mode. Il est important de noter qu'une constante d'un mode d'union appartient en fait toujours au mode composant l'union auquel il appartenait avant. L'appartenance d'une valeur à un mode n'est pas perdue dans l'union.

La coercition "uniting" nous le montre :

Exemple : union (real, bool)a:= true
true sera converti du mode bool au mode union (real, bool)
mais le langage ne perd pas de vue que true est booléen.

2.3. Tableau récapitulatif

Nous avons rassemblé tout ce qui précède dans un tableau d'ensemble (T.III.A.1.). Nous nous sommes surtout attachés à y montrer les objets nommés et/ou nommables, puisque c'est là notre objectif principal. De plus, nous y avons fait entrer la notion de mode de base.

> Un mode est de base si et seulement si
> il est défini et non définissable.

Ce sont donc bien les modes que le programmeur n'a pas la possibilité de définir lui-même et qui servent de base à la construction des autres.

Un seul domaine a, jusqu'ici, échappé à notre classification : les "labels". Nulle part, ni dans R. ni dans I.I. on ne précise quelle est la valeur nommée par un label. Nous ne pouvons donc que conjecturer.
Un "label" peut être considéré comme le nom d'un "jump point" et, dans ce cas, on placerait, comme en Algol 60, le domaine des "jumps points" parmi les domaines de base. Il y a cependant là encore matière à discuter car les "jump points" ne seraient-ils pas définissables plutôt que définis ? Rappelons toutefois que le langage ne précise absolument rien à ce sujet.

Nous employons les abréviations suivantes dans le tableau :
di = défini(s), da = définissable(s), né = nommé(s), na = nommable(s), ex = exemple, x = oui, - = non.

T.III.A.1.

Domaines	Eléments di	da	né	na	ex.(dénotation)	Modes di	da	né	na	nom	Opérateurs di	da	né	na	ex.
entiers	x	-	x	x	12	x	-	x	x	int (1)	x	-	x	x	+
réels	x	-	x	x	3.14	x	-	x	x	real (1)	x	-	x	x	+
booléens	x	-	x	x	false	x	-	x	x	bool (1)	x	-	x	x	∨
caractères	x	-	x	x	"a"	x	-	x	x	char (1)	x	-	x	x	ne
formats	x	-	x	x	$ 3d $	x	-	x	x	format (1)	-	-	-	-	
bits	x	x	x	x	101100101	x	x	x	x	bits (1)	x	-	x	x	not
bytes	x	x	-	x		x	x	x	x	bytes (1)	x	-	x	x	ge
strings	x	x	x	x	"ab.cd"	x	x	x	x	string (1)	x	-	x	x	⩽
sémaphores	x	x	-	x		x	x	x	x	sema (1)	x	-	x	x	up
complexes	x	x	-	x		x	x	x	x	compl (1)	x	-	x	x	re
routines	x	-	x	x		x	x	x	x	proc	-	-	-	-	
multiples	-	x	-	x		-	x	-	x		x	-	-	x	
structures	-	x	-	x		x	x	x	x	file, sema..	x	-	-	x	
union	-	x	-	x		-	x	-	x		x	-	-	x	
références autres que proc.	-	x	-	x		-	x	-	x		x	-	-	x	
procédures	-	x	-	x		-	x	-	x		-	-	-	-	
actions primitives	x	-	-	-		-	-	-	-		-	-	-	-	
labels			-	x		-	-	-	-		-	-	-	-	

Dans les cas où nous représentons les caractéristiques d'une famille d'objets par le diagramme suivant :

di	da	né	na
x	x	x	x

nous entendons que les objets en question sont tous définissables et nommables, mais que seulement certains d'entre eux sont définis et nommés. Il y a cependant deux exceptions à ce principe, les bits et les strings qui possèdent tous les quatre propriétés.

Dans tous les autres cas, il s'agit toujours de tous les objets du domaine analysé qui sont définis, nommés, définissables ou nommables.
Le chiffre 1 entre parenthèses à côté des noms de mode signifie que, pour le domaine considéré, il n'existe qu'un seul mode.
La dernière colonne indique s'il existe des opérateurs définis sur le domaine étudié. On y trouvera "-" pour toute la colonne "da" car les opérateurs appartiennent aux domaines de base.

B. Relation nom-objet-valeur

En Algol 68, les relations entre nom, objet et valeur s'appuient sur le mécanisme des déclarations. C'est pourquoi nous allons tout d'abord essayer de dégager la signification de ce mécanisme en Algol 68.

1. Mécanisme des déclarations

Le rôle d'une déclaration est de donner une définition : telle "dénomination" désigne tel objet. Par "dénomination", on entendra généralement identificateur. Il existe trois sortes de déclarations en Algol 68 : les déclarations de mode, celles d'objets appartenant à un mode et celles de priorités.
Exemples :

real e = 2.7182 signifie: "e" désigne le réel 2.7182

node book = struct (string title, ref book next)
signifie:"book" désigne le mode défini par "struct (string title, ref book next)" (qui est un déclareur)

priority + = 6 signifie:la priorité de + est désignée par "6".
(Le schéma est légèrement différent ici mais le principe reste le même).

Une déclaration permet donc de donner un nom à des objets. Mais quels sont les objets intervenant en Algol 68 ?
"Un programme Algol 68 peut être analysé sous la forme d'un arbre composé d'"objets externes" (tels que des identificateurs, des opérateurs, des dénotations, des indications, des phrases....). En tant que tel, il définit par élaboration une séquence d'"actions" dans un "calculateur" (que celui-ci soit un être humain ou un automate). Ces actions sont exécutées sur des "objets internes", quelque part dans l'esprit humain ou dans la mémoire de l'automate.
Chaque objet interne a les trois propriétés suivantes : il appartient à un certain mode, il est une instance particulière d'une valeur (ou objet) de ce mode et il a une certaine location." (I.I. 1.1.1.)

On remarque qu'en Algol 68, la location est associée à l'objet et non plus à son nom.
Les objets externes sont en fait des "phrases", c'est-à-dire "un morceau de texte qui spécifie une action". Il en existe de trois sortes, les expressions qui produisent une valeur, les déclarations et les instructions. Parmi ces objets externes, les seuls qui produisent une valeur sont les expressions. Elles ont donc un mode, celui de la valeur produite. Il découle de là que seuls les cas simples d'expressions, c'est-à-dire les identificateurs et les dénotations, sont susceptibles d'être directement reliés à des objets internes.

"Une dénotation est "une production terminale d'une notion" dont la valeur est indépendante de l'élaboration du programme." (R.5) C'est ce qu'on appelle généralement "littéraux" ou "constantes". Exemple: 3.14 ou 'abc'. En fait, il s'agit des noms standard des objets appartenant aux domaines de base. Ce sont donc les noms de ce que nous avons appelé les objets nommés. Par opposition, les identificateurs sont les noms que le programmeur définit lui-même et qui n'existent que pour un programme donné. Ce sont les noms éventuels des objets nommables. A une dénotation ne correspond jamais qu'un seul et unique objet, et ce de façon immuable.

2. Relations objets externes-objets internes

Il existe deux relations entre objets, la première entre objets externes et objets internes, celle de "possession" et la deuxième entre objets internes, celle de "repérage".

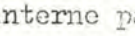
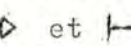
Un objet externe possède un objet interne, celui dont il est le nom. Cette relation de possession est établie soit par définition dans le langage si l'objet externe est une dénotation, soit par déclaration dans un programme, s'il s'agit d'un identificateur. Puisque l'identificateur n'a qu'une existence limitée, il en est de même pour la relation de possession entre cet identificateur et un objet interne.

Exemples : 3.14 est une dénotation qui possède le réel nommé par 3.14.

proc p = (int i) real : if i = o then i+1 else i-1 fi;

p est un identificateur qui possède la routine à un paramètre entier et à résultat réel, définie par le membre de droite de la déclaration (donc en fait, la routine possédée par la dénotation figurant à droite).(R.5.4)

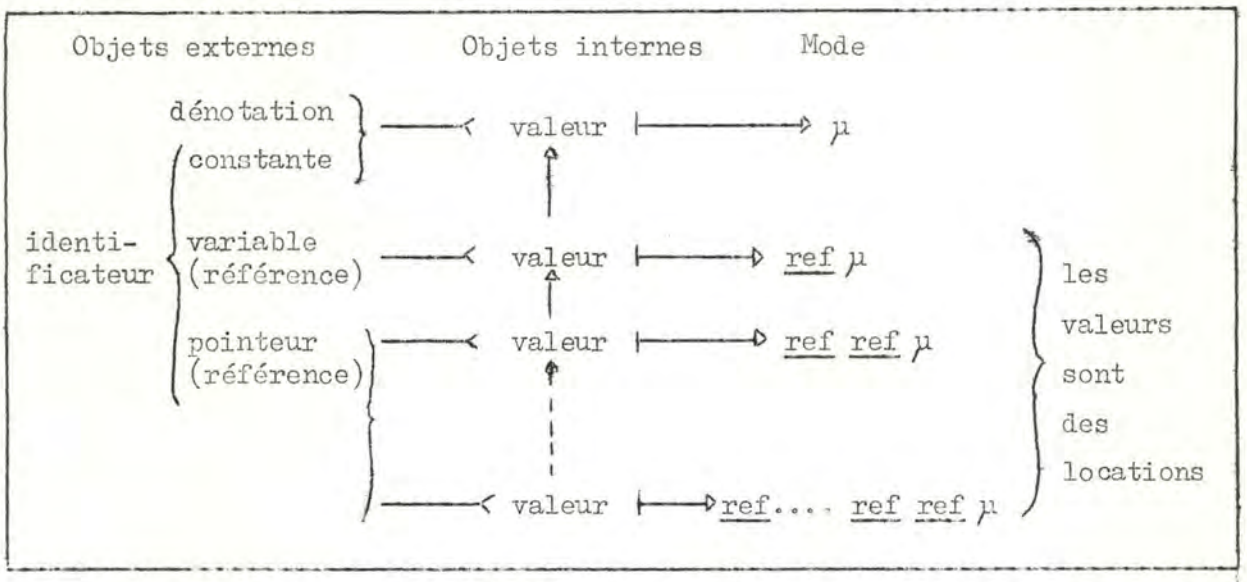
Pour qu'un objet interne repère un autre objet interne, il faut que le premier soit une référence à un mode μ et le second un objet de mode μ ou, ce qui revient au même, que le premier soit un objet de mode ref μ et le second de mode μ. Dans ce cas, la location du second objet est à la disposition du programmeur sous la forme d'une référence.

Nous allons maintenant examiner ces relations pour les différents domaines rencontrés en Algol 68. Nous allons utiliser les conventions suivantes dans ce qui suit : μ représentera toujours un mode pour lequel il n'existe pas de mode μ' tel que μ soit le mode ref μ'; le mot "valeur" désignera un objet interne particulier appartenant à un mode; les notations  ;  et signifient respectivement "possède", "repère" et "appartient à".

2.1. Domaines de base et références

On constatera, du fait des relations de repérage, que les locations ne sont pas indépendantes. Cela ressortira encore plus clairement des diagrammes B.2.1, B.2.4 et B.2.6.

- Opérandes



B.2.1.

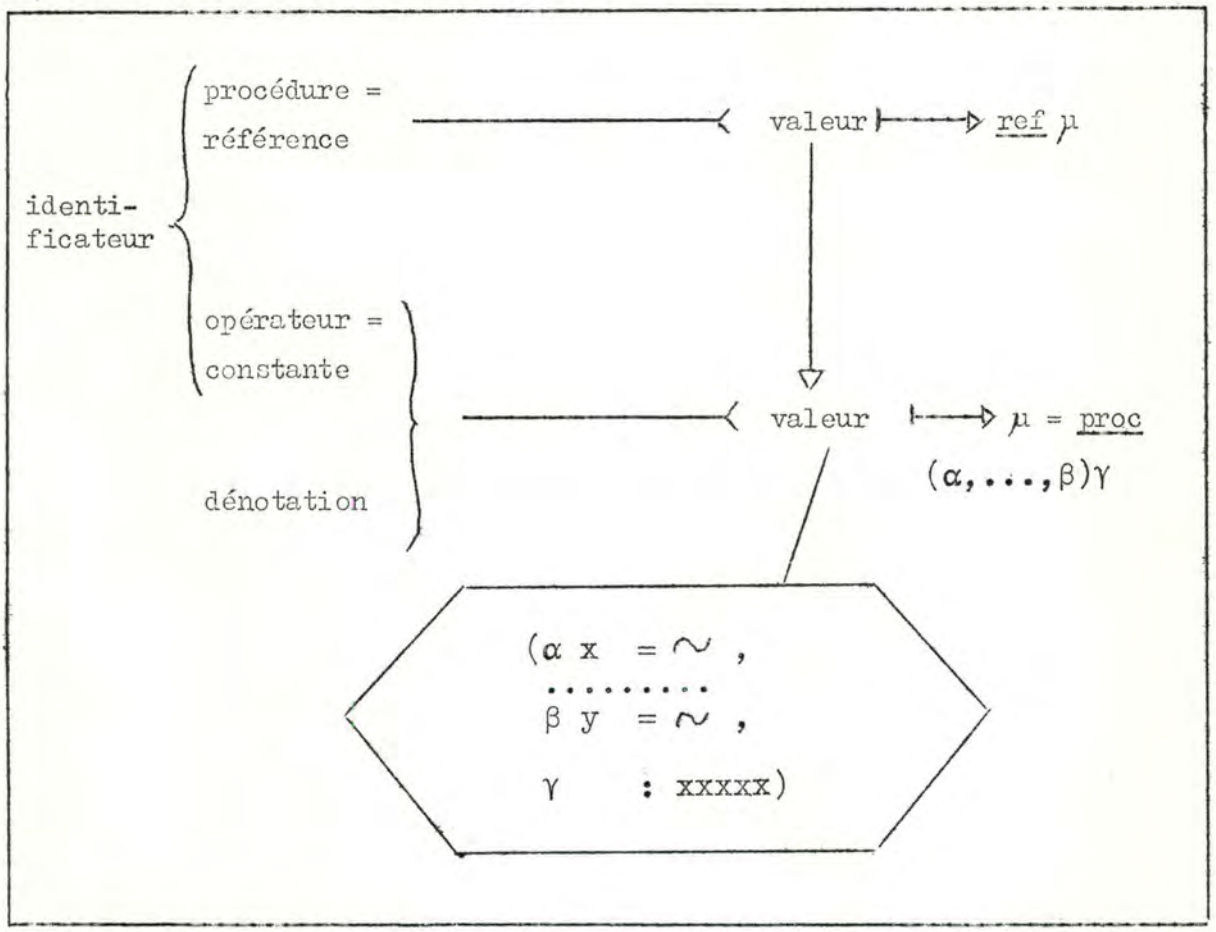
Dans le diagramme ci-dessus, μ est soit real, int, bool, char, format, bits ou string, c'est-à-dire les modes de base ou assimilés pour lesquels il existe des dénotations.
Pour les domaines assimilés aux domaines de base, mais n'ayant pas de dénotations, un schéma semblable reste encore valable, en y supprimant les dénotations. Il s'agit des bytes et des sema (avec quelques restrictions pour ce dernier car mode sema = struct (ref int)!).

- Opérateurs

Il existe des dénotations de routines en Algol 68 (R.5.4). Ces dénotations, objets externes, possèdent les routines (objets internes) correspondantes. (R.5.4.2). Le procédé est donc bien similaire à celui des opérandes de base. Les autres objets externes qui correspondent à des routines sont les identificateurs de procédures et d'opérateurs.
Il existe cependant une différence fondamentale entre les opérateurs et les procédures : les opérateurs sont des constantes de mode procédure et ne peuvent jamais être des références. En effet, la valeur possédée par un opérateur, donc la routine possédée, dépend très souvent du contexte, ce qui rend le mécanisme de référence impossible (R.4.3.). Par contre, les procédures, quant à elles, sont toujours des références (II.4.2.1.).
En fait, un mode " procédure " est un mode composé de routines

à n paramètres, n éventuellement nul, et avec résultat d'un certain mode, celui-ci pouvant être vide. Les constantes d'un tel mode sont des routines, tandis que les variables sur ces modes sont des procédures.

On a donc le diagramme B.2.2.



B.2.2.

x et y représentent des identificateurs, ~ des paramètres actuels et xxxxx le corps de la routine.

Que se passe-t-il maintenant lors d'un appel de procédure ou de l'emploi d'un opérateur ?

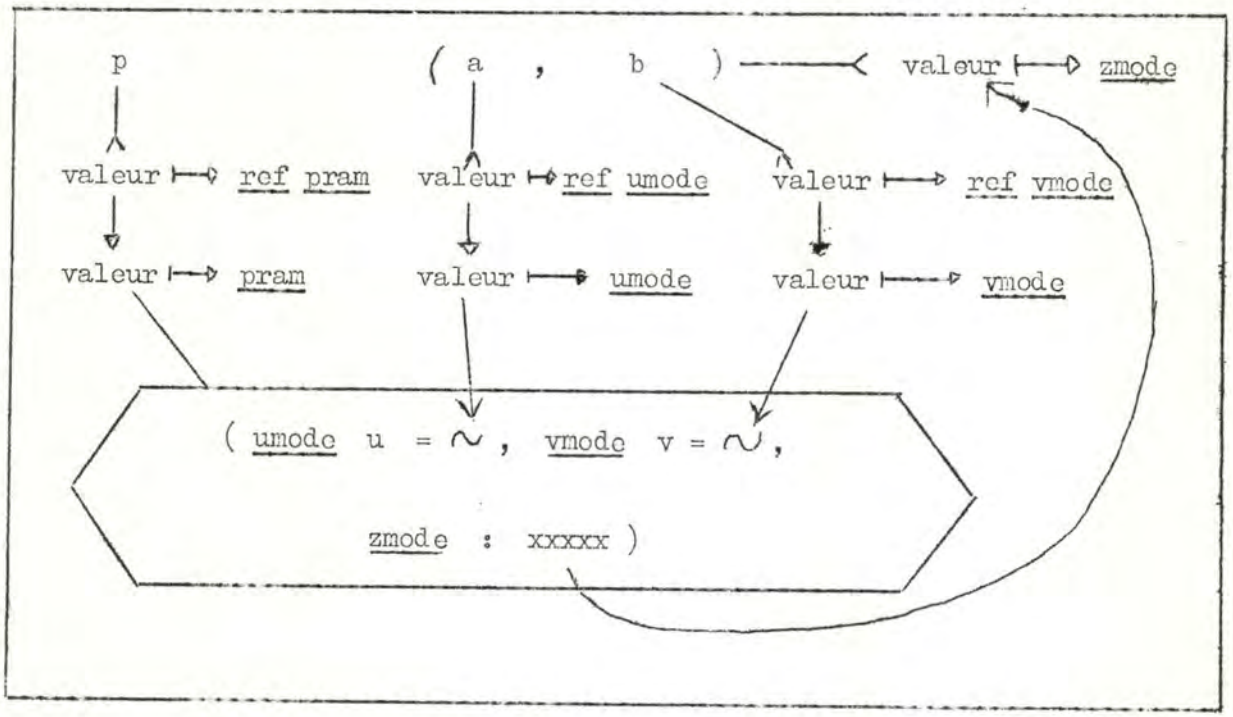
Soit les déclarations suivantes :

mode pram = proc (umode, vmode) zmode

ref pram p = ((umode u, vmode v) zmode : xxxxx)

où xxxxx représente le corps de la procédure.

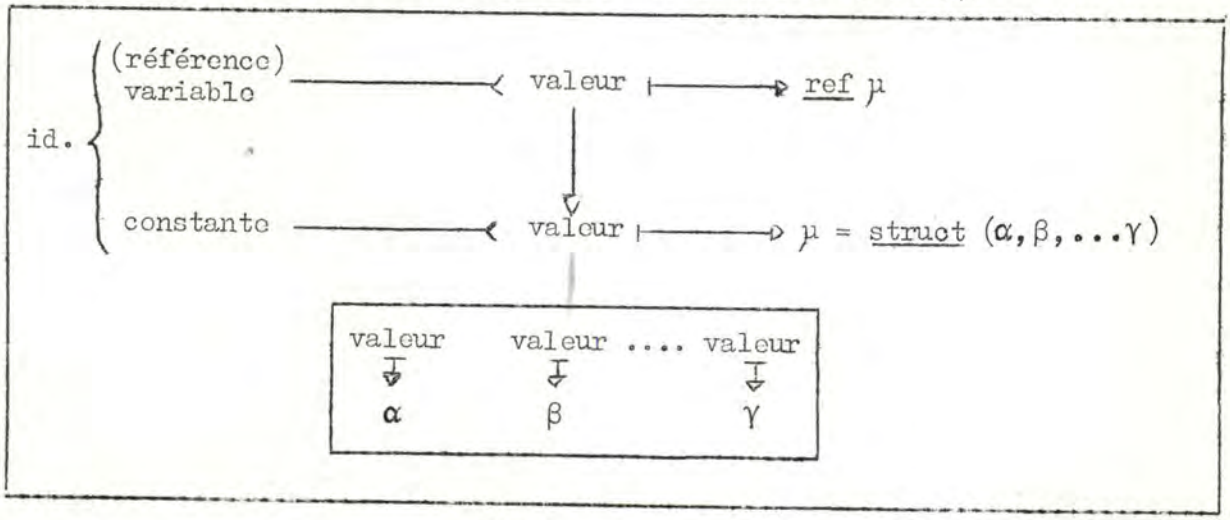
Le diagramme B.2.3. donne les relations entrant en jeu si l'on a p(a,b). Le schéma serait semblable pour un opérateur !



B.2.3. où " ⟶ " (différent de " ⟼ ") indique l'action de copie, c'est-à-dire : l'objet à l'extrémité droite de ⟶ est une nouvelle instance de l'objet à l'extrémité gauche.

2.2. Domaines dérivés, références non comprises

- Structures : les relations sont schématisées en B.2.4.

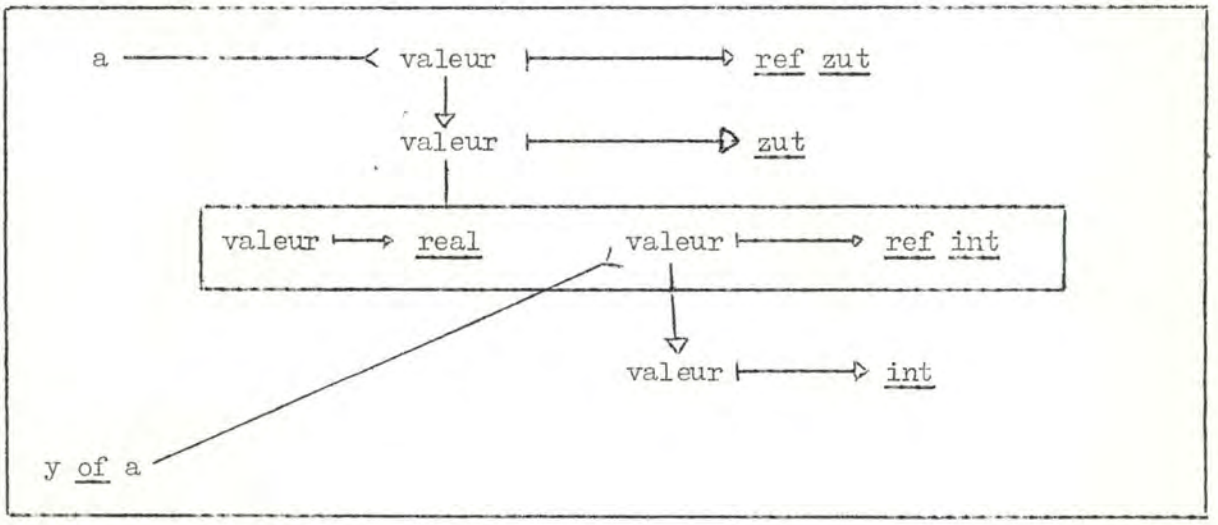


B.2.4.

Il est important de noter que les sélecteurs de "fields" ne sont pas des identificateurs et donc pas des références !
La valeur ou objet interne de mode μ se compose de différentes valeurs de modes α, β, ... γ. Que se passe-t-il quand on nomme un "field" (par coercition) ?
Supposons que l'on ait défini le mode zut :

mode zut = struct (real x, ref int y)

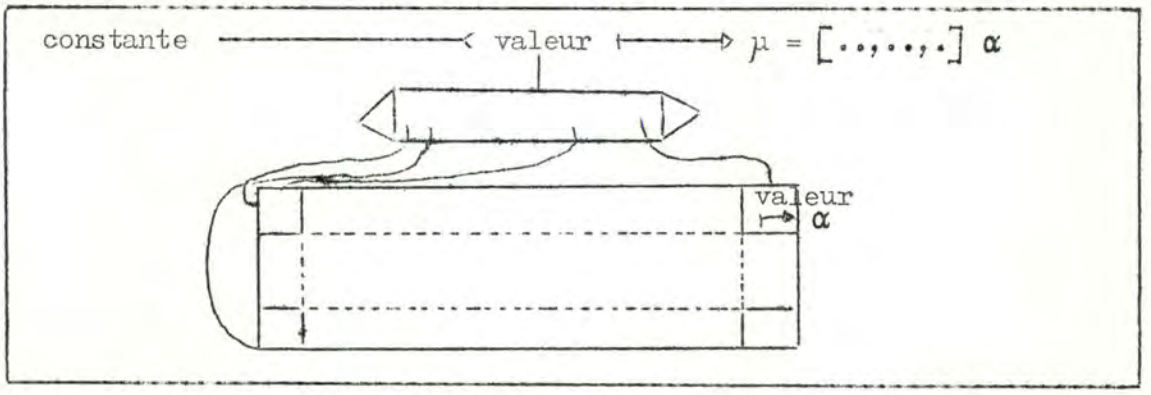
et que l'on déclare ref zut a;
Quelles sont les relations de possession et de repérage qui entrent en jeu lors de l'emploi de y of a ? C'est ce que nous montre le diagramme B.2.5.



B.2.5.

y of a désigne une référence, mais y n'est pas un identificateur.

- Multiples : voir schéma B.2.6.



B.2.6.

Nous y avons supprimé le niveau des références uniquement pour ne pas nous répéter, mais il est certain que cela n'empêche pas son existence. Le descripteur (des bornes) du multiple fait partie de sa valeur.

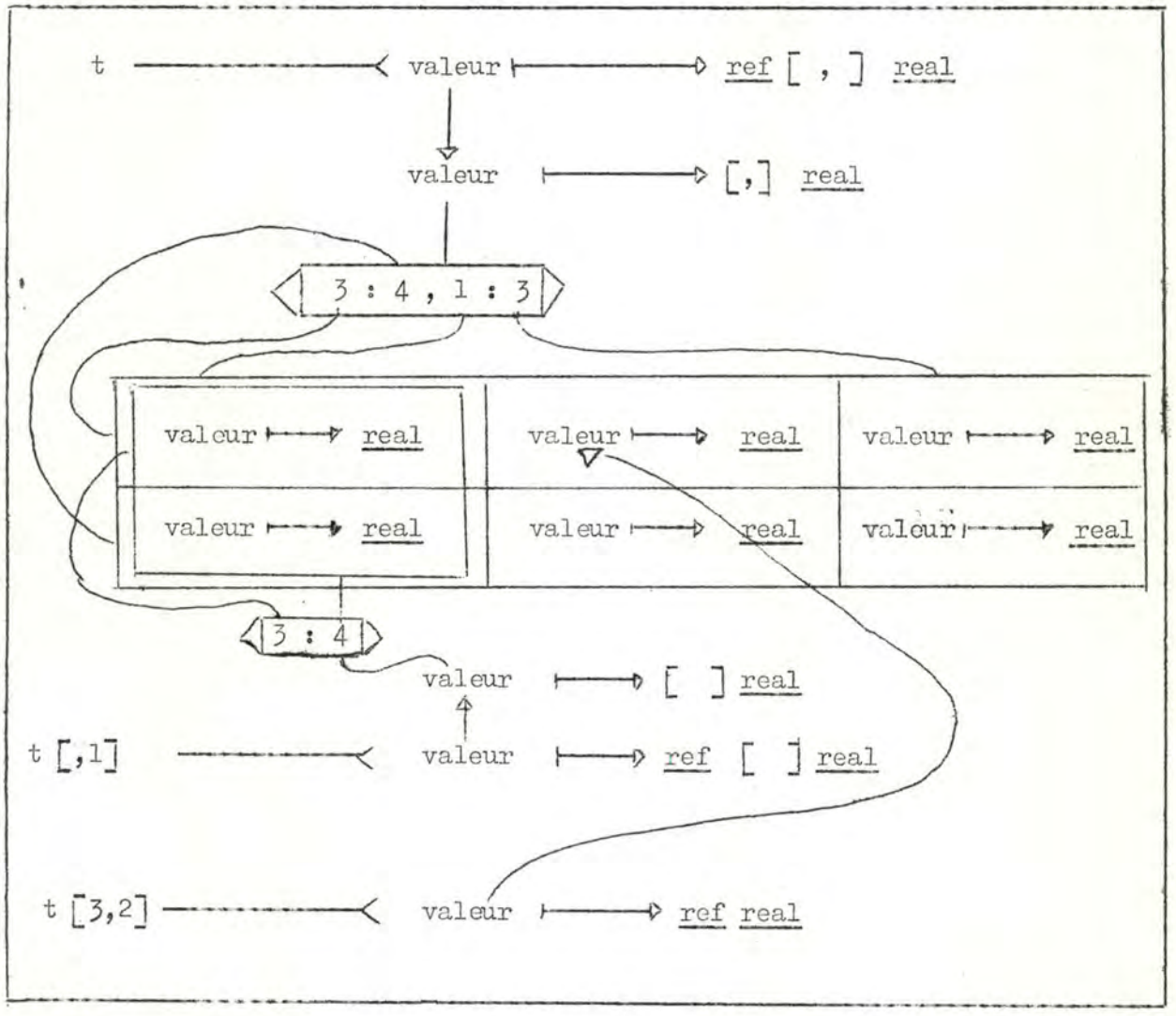
Supposons que l'on ait la déclaration suivante :

ref [3:4 , 1:3] real t

t [3,2] est de mode ref real et

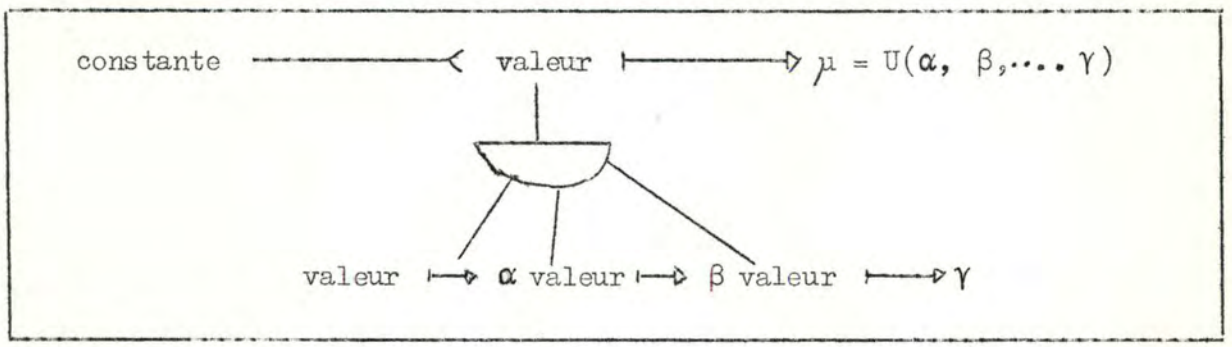
t [,1] de mode ref [ ] real.

On a alors le diagramme B.2.7.



B.2.7.

- Union : dans le diagramme B.2.8., ⌓ signifie qu'il y a choix d'une et une seule des différentes valeurs qui sont reliées à ce symbole.



B.2.8.

Chapitre Quatre

ANALYSE GENERALE

Nous nous proposons, dans ce chapitre, de reprendre différents sujets de réflexion que nous avons abordés dans les chapitres précédents. Nous comparerons ainsi les deux manières de considérer l'assignation proposées par Strachey et dans Algol 68. Nous analyserons ensuite le problème des relations entre noms, objets et valeurs. Nous terminerons enfin par l'étude des distinctions à établir entre opérandes et opérateurs.
Nous pourrons alors, sur base de cette analyse, établir une classification des objets nommables, ce que nous ferons dans un chapitre ultérieur.

A. Les deux aspects de l'assignation

Si l'on essaie de découvrir le sens exact de l'assignation, on se trouve confronté à deux optiques relativement différentes : celle de Ch. Strachey, qui a été appliquée à la conception de B.C.P.L. et de C.P.L. et celle introduite par Algol 68.
Si l'on écarte Algol 68, les langages algorithmiques où l'on rencontre l'assignation ont tous suivi l'optique de Strachey, sans toutefois aller toujours jusque dans ses conséquences au point de vue $\mathcal{L}$ - et $\mathcal{R}$-valeurs.

Nous allons essayer de montrer quels sont les fondements de ces deux points de vue.

Du point de vue de l'exécution, le différend n'existe pas. En effet, l'expression (généralement seul un identificateur sera admis) de gauche doit être d'un mode tel qu'elle puisse désigner des valeurs appartenant au même mode que l'expression de droite. Si nécessaire, cette dernière subira une conversion de mode généralement implicite afin de permettre l'affectation. Il y aura donc évaluation séparée des deux membres suivie des conversions si nécessaire, puis affectation de la valeur de l'expression de droite à celle de gauche.
Algol 68 ajoute à cela que l'assignation a un résultat qui est la valeur du membre de gauche après l'exécution.

La différence entre les deux optiques porte en fait sur la nature de l'objet nommé par ":=" dans la plupart des langages.
Algol 68 est le seul, à notre connaissance, à considérer ":=" comme le nom d'une opération avec opérandes et résultat.
Par contre, on considère généralement, et c'est le cas de Strachey, que l'assignation joue un rôle particulier parce que son exécution modifie le contenu de la mémoire. Un opérateur, au sens habituel du terme, effectue une certaine action sur des opérandes et produit un résultat mais il n'y aura changement de l'état de la mémoire que lors de l'assignation de ce résultat. Donc l'assignation, dans cette façon de voir, est une commande et non un opérateur.

Il est cependant intéressant de remarquer que, pour formuler une telle opinion, il a fallu analyser le rôle de l'assignation dans un programme déjà compilé. En effet, si l'on se situe au niveau du langage proprement dit, donc avant toute compilation, il nous semble que l'on ne pourra pas conserver cette façon de voir, et cela pour deux raisons qui nous paraissent essentielles.

- Situons-nous au niveau du langage et donc avant la compilation. Parmi les actions qui ne sont pas considérées en général comme des opérateurs (sauf éventuellement en Algol 68), certainesprovoquent une modification de l'état de la mémoire. Songeons aux déclarations de variables, de constantes et de modes principalement (modification d'une table de symboles, etc...). Ceci enlève donc une grande part de la particularité habituellement attribuée au rôle de l'assignation.

- De plus, on peut aisément se rendre compte que les opérateurs proprement dits modifient l'état de la mémoire. Ils provoquent la mise à 0 ou 1 de codes conditions, la modification de certains registres ou de positions privilégiées de la mémoire. Or, au niveau du langage, il est impossible de savoir si les registres de travail et le ou les accumulateur(s) se trouvent ou non en mémoire centrale puisque cela dépend des machines.
De plus, c'est de peu d'importance : ce sont de toute façon des "endroits" où des valeurs seront stockées et ils font donc partie de la mémoire prise en un sens général.

Les opérateurs provoquent tous une modification de l'état de la mémoire puisque le résultat de l'opération doit bien être mis quelque part avant d'être assigné.

Outre le fait qu'analyser les propriétés d'un langage avant toute compilation revient à se situer réellement au niveau de celui-ci, il est essentiel de le faire. En effet, la distinction entre "compilation" et "exécution" est parfois très nette (pratiquement toujours dans les langages de programmation évolués, sauf pour de rares exceptions telles que LISP), mais est généralement artificielle et parfois sans signification.

De tout ceci, il résulte qu'il nous semble plus normal de considérer l'assignation comme une opération et non pas comme une commande particulière.

Les déclarations, quant à elles, sont donc aussi des opérations qui créent une relation entre un nom et un objet. C'est à ce point essentiel dans un langage que là où les déclarations explicites ne sont pas obligatoires, la définition du langage prévoit des déclarations implicites par une série de conventions.

Exemple : tout identificateur commençant par I, J, K, L, M ou N, en Fortran, sauf déclaration explicite d'appartenance à un autre mode, désigne une variable de type entier.

Les différentes opérations qui agissent sur les relations "possède" et "repère" sont donc les suivantes :

1) les déclarations qui créent des relations "possède" entre des noms et des objets internes,
2) l'assignation qui crée et modifie des relations "repère" entre des objets internes,
3) par extension, les fins de blocs et de programme qui suppriment des relations "possède" entre noms et objets (et donc aussi les relations "repère" entre ces objets internes et d'autres),
4) les routines qui contiennent des déclarations, assignations et éventuels effets de bord.

B. Relation non-notion-attribut

Nous n'avons pas cessé, dans l'ensemble de ce travail, de parler de "nommer des objets", de "relation entre un nom et sa valeur", de "relation entre un nom et l'objet qu'il désigne",... Il nous semble qu'il serait maintenant nécessaire de préciser ce que ces expressions signifient réellement dans le cadre d'un langage.

Un langage est utilisé pour parler d'un certain univers. Pour les langages naturels, il s'agit du monde matériel qui nous entoure. Il faut cependant remarquer que les langages ne parlent pas directement des objets de cet univers, mais seulement de la façon dont on les considère à un moment donné, c'est-à-dire des notions que nous avons relativement à cet univers.

Nous sommes donc amenés à considérer trois ensembles : les phrases du langage, l'univers dont le langage permet de parler et les notions qui constituent la façon dont nous voyons l'univers. La structure propre à l'ensemble des phrases du langage constitue la syntaxe de celui-ci et l'ensemble des relations unissant l'ensemble des phrases du langage à l'ensemble des notions constitue la sémantique de ce langage.

Si l'on se pose la question de savoir ce qu'est une notion, nous nous apercevons qu'elle est définie par un ensemble d'attributs. Exemple: s'il s'agit d'un objet, les attributs seront la forme, la couleur, l'usage, la consistance, etc... Le problème revient à donner une définition de l'attribut. En accord avec C. Cherton, nous définirons l'attribut comme un couple ordonné: la nature d'attribut et la valeur d'attribut. Exemple: attribut de nature couleur et de valeur bleu.
Les natures d'attribut sont elles-mêmes des notions. Quant aux valeurs d'attribut, elles peuvent être elles aussi des notions mais peuvent aussi être un objet de l'univers.

Limitons-nous maintenant au problème de la nommabilité dans un langage, c'est-à-dire aux phrases du langage qui sont des identificateurs et des dénotations.

En vertu de ce qui précède, les "choses nommables" que nous étudions sont en fait les "notions nommables". Une déclaration crée donc une relation entre un nom et une notion.
Nous allons voir que les relations "nom-objet-valeur" ne sont en fait qu'un cas particulier de relations "nom-notion-attribut". "Valeur" est une notion qui joue un rôle particulier dans les langages de programmation, tout comme "mode", "real", etc...
Prenons un exemple :

Une variable est une notion. Parmi les attributs de cette notion, il en est deux qui jouent un rôle intéressant pour nous : l'attribut de nature "mode" dont la valeur pourra être real, boolean, etc... (donc une autre notion) et l'attribut de nature "valeur" dont la valeur sera un réel, booléen, etc... donc une notion qui a une "représentation" dans l'univers dont le langage permet de parler.

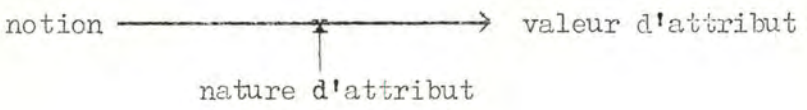
En Algol 68, nous avions deux relations différentes entre objets : "possède" entre objets externes (noms) et objets internes et "repère" entre objets internes.

Dans le cadre plus général des notions, la première sera remplacée par la relation "désigne" qui existe entre un nom et la notion qu'il nomme et la seconde n'est rien d'autre que celle qui existe entre une notion et sa valeur d'attribut pour l'attribut de nature "référence".

Un des résultats qui nous semble très important, à ce niveau, est que nous n'avons plus aucun besoin de parler de location et donc de nous soucier à quelque degré que ce soit de l'inplémentation. Il nous semble en effet gênant de devoir parler de la location d'un objet comme d'une de ces propriétés essentielles. Cela revient en effet à considérer que l'on a besoin de la machine pour comprendre ce qu'est un objet traité par un langage!

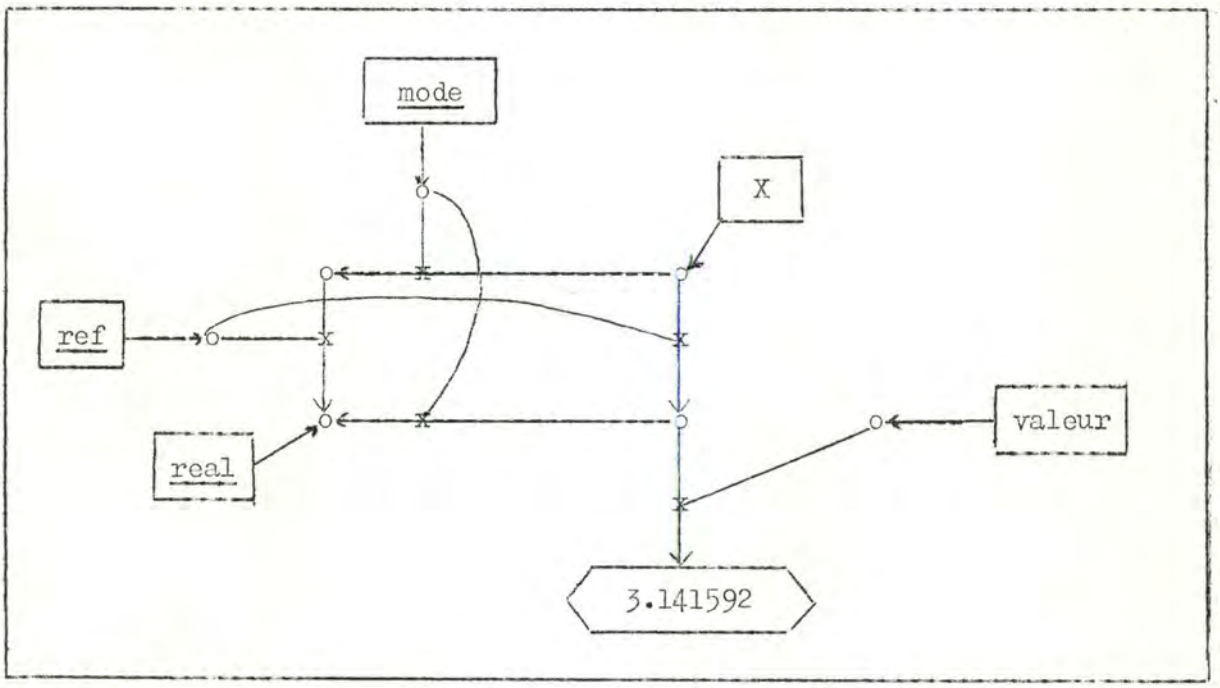
Afin de clarifier ce qui précède, nous allons illustrer dans les diagrammes suivants les relations entre notions du genre "référence", "structure", "procédure", "tableau", etc... rencontrées en Algol 68. Une rapide comparaison entre les diagrammes du paragraphe III B et ceux qui vont suivre montrera que les relations "nom-objet-valeur" sont bien un cas particulier

des relations "nom-~~objet~~ notion-attribut". Nous utiliserons pour ce faire le symbolisme proposé par Cherton dans "Essai de formalisation de la sémantique" (voir Bibliographie). Les notions seront représentées par des petits ronds (o), les relations entre une notion, une nature et une valeur d'attribut par des "flèches" disposées comme suit :



Les objets d'univers qui interviendraient figureront dans une cartouche à bouts pointus (⬡) et les noms du langage dans des cartouches rectangulaires (▭). Enfin, ces noms seront reliés à la notion qu'ils nomment par une flèche ⟶ (qui représente donc la relation "désigne").
Nous avons pris comme mode de base du genre opérande le mode real. Toutefois, les mêmes diagrammes seraient valables pour les autres modes de base.

- soit la déclaration ref real X ;
  et X := 3.141592;
  cela nous donne le diagramme B.1.



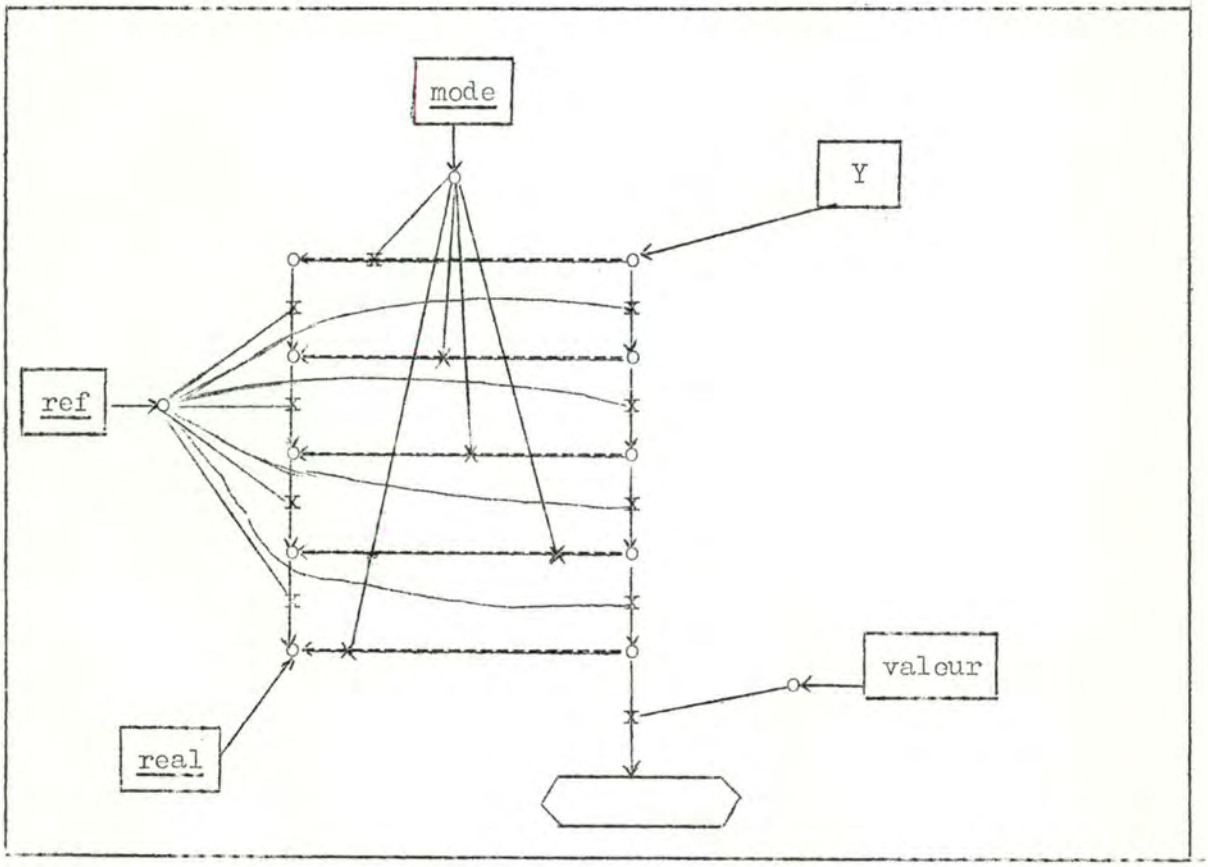
B.1.

"valeur" est une nature d'attribut, mais cette notion n'est pas nommée en Algol 68, c'est pourquoi le mot "valeur" n'est pas souligné.

Supposons maintenant que l'on ait :

ref ref ref ref real Y

On obtient le diagramme B.2. qui est simplement le diagramme B.1. avec quelques "niveaux de référence" supplémentaires.



B.2.

- structures : mode flûte = struct (real re, int in, ref bool b, zut x)

où zut est un mode défini ailleurs dans le programme.

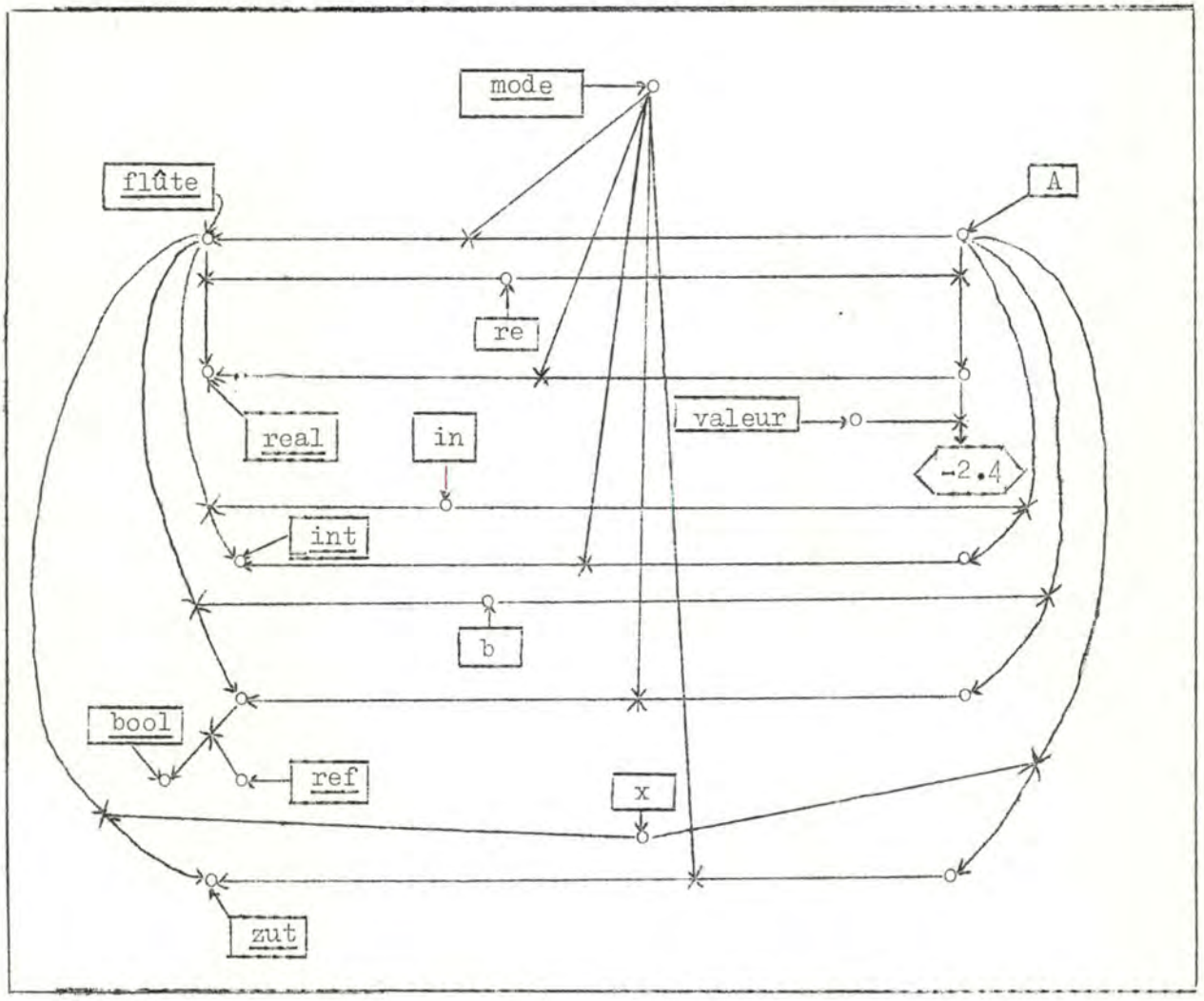
Soit alors flûte A ;

re of A := -2.4;

Nous obtenons alors le diagramme B.3.

Le schéma devient de suite plus complexe. On y remarquera que les sélecteurs sont en fait des natures d'attribut. C'est ainsi que re of A désigne la notion qui est valeur de l'attribut de la notion A dont la nature est re.

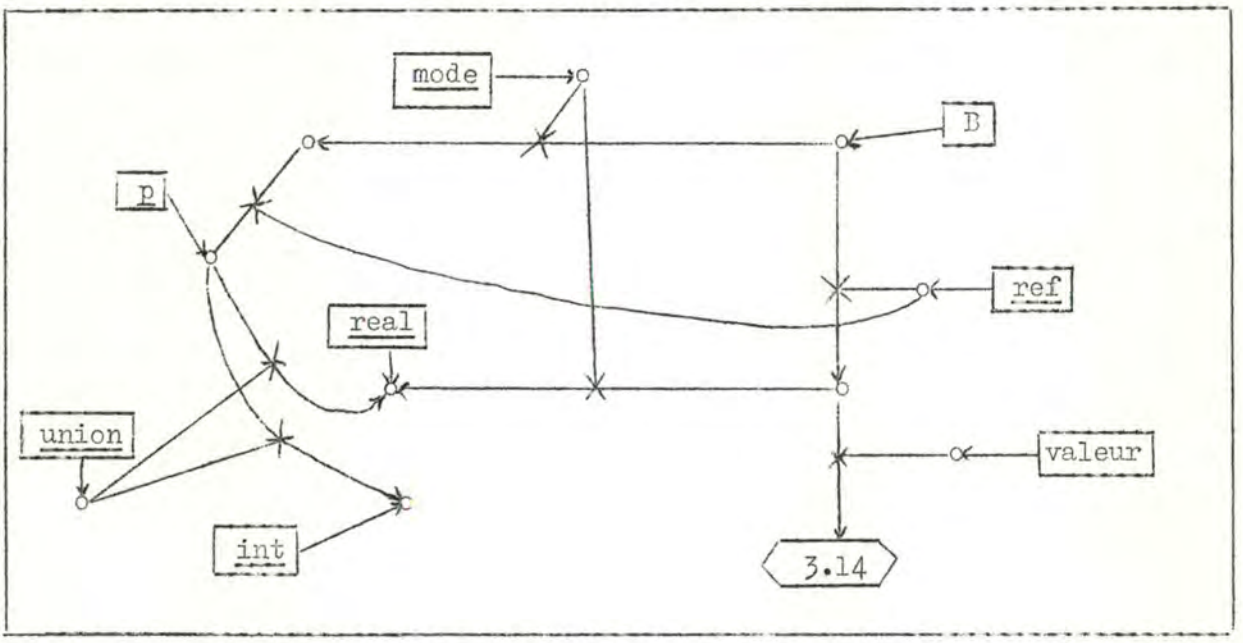
Si l'on s'intéresse au mode de cette notion, on aura la valeur de l'attribut de nature mode real, et si l'on s'intéresse à sa valeur, on obtient l'objet -2.4, valeur de l'attribut de nature "valeur".



B.3

Nous n'avons tenu compte de l'attribut de nature "valeur" que pour le premier "field", il est évident qu'on peut procéder de même pour chacun des "fields".

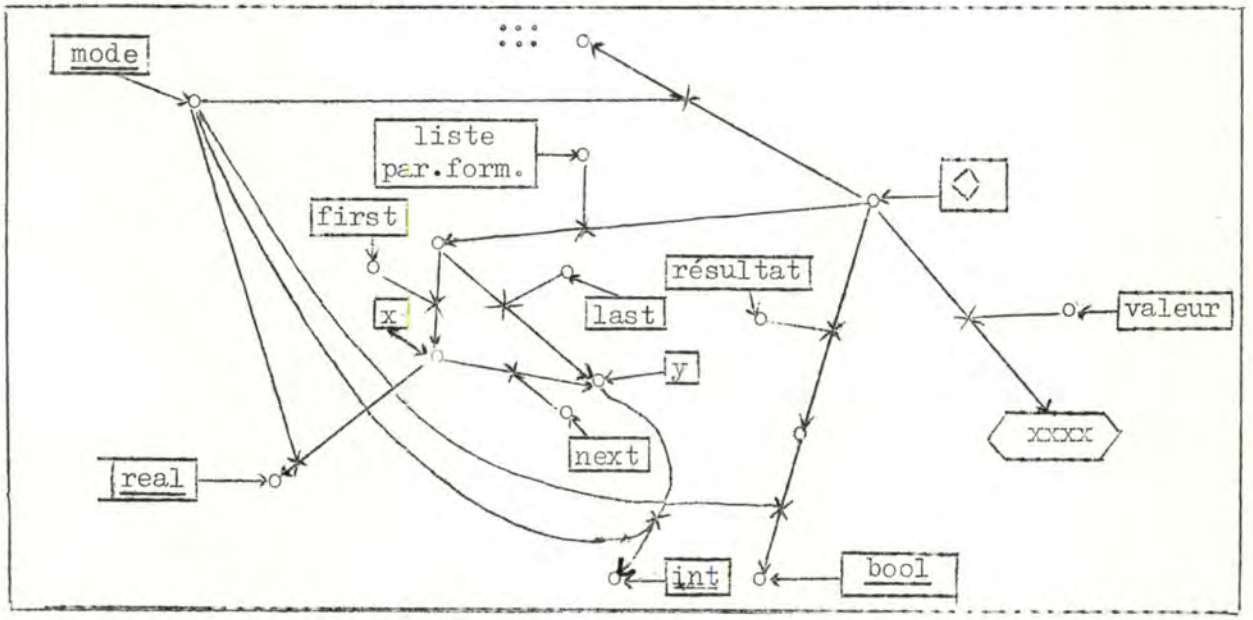
- union : mode p = union (int, real);
  ref p B ; B := 3.14



B.4

La notion repérée par B a un attribut "mode" dont la valeur est soit real soit int, mais jamais les deux!

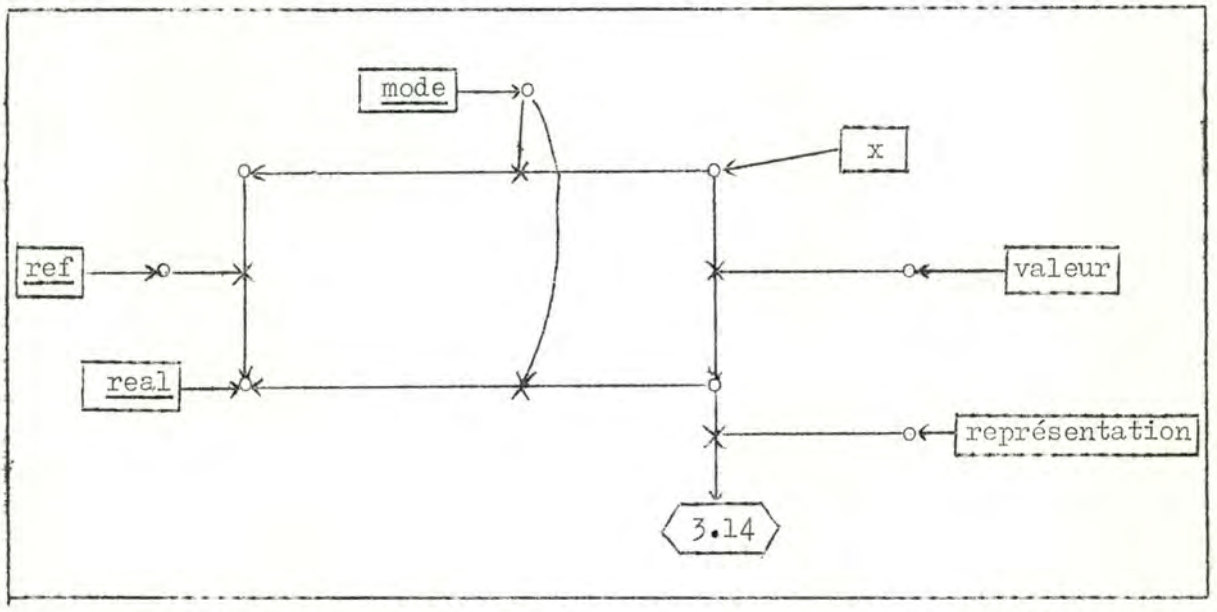
- routine : op ◇ = ((real x, int y) bool : xxxx)



B.5

Il est très important de noter que les attributs qui définissent une notion dépendent de la façon dont on s'intéresse à cette notion. C'est ainsi que, sous l'optique Algol 68, nous avons pu construire les diagrammes B 1 à 5.

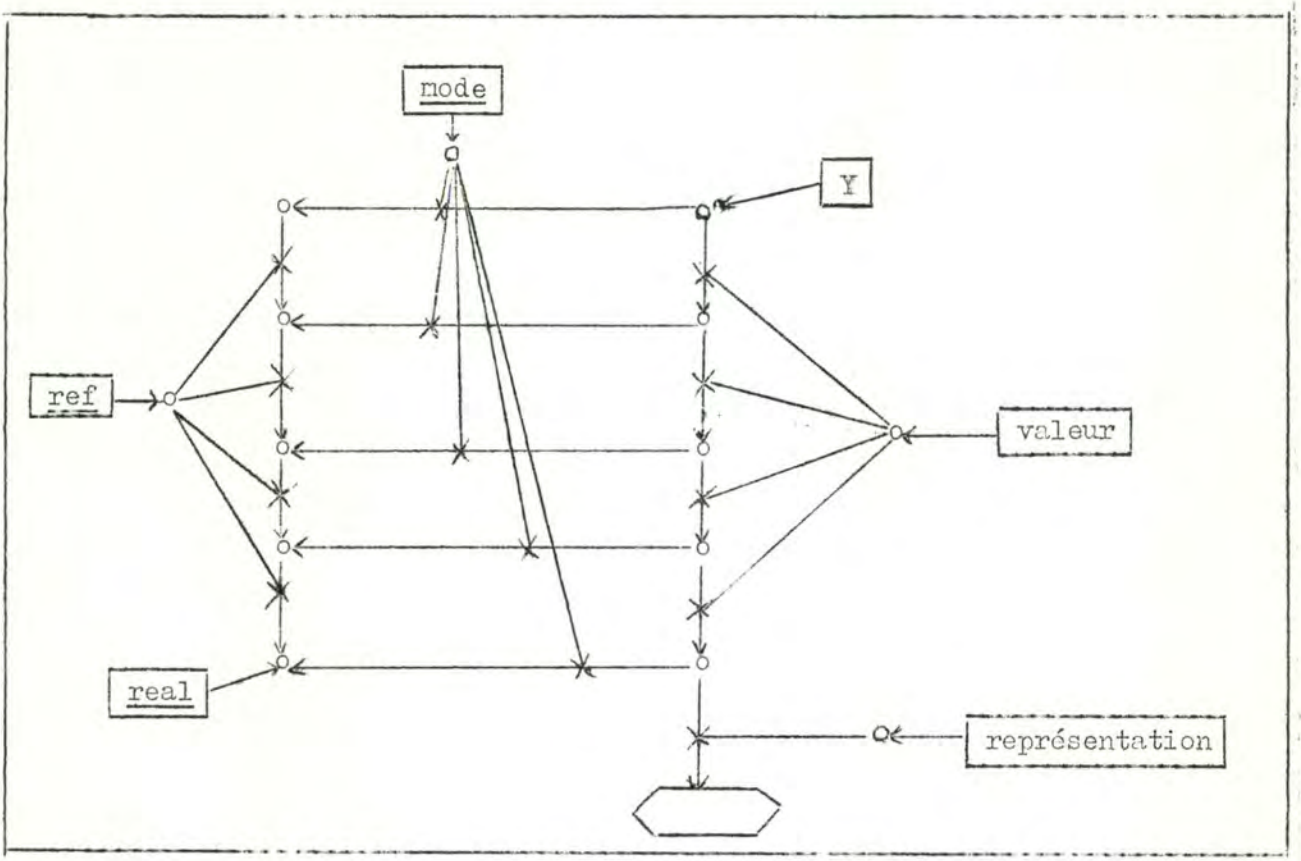
Si nous reprenons maintenant notre optique à propos des langages en général, (voir chapitre un), nous voyons qu'il existe des notions qui ont un attribut "représentation" dont la valeur sera toujours un objet de l'univers considéré par le langage étudié. Le diagramme B.1 pourrait, dans notre façon d'interpréter les choses, donner ceci :



B.6

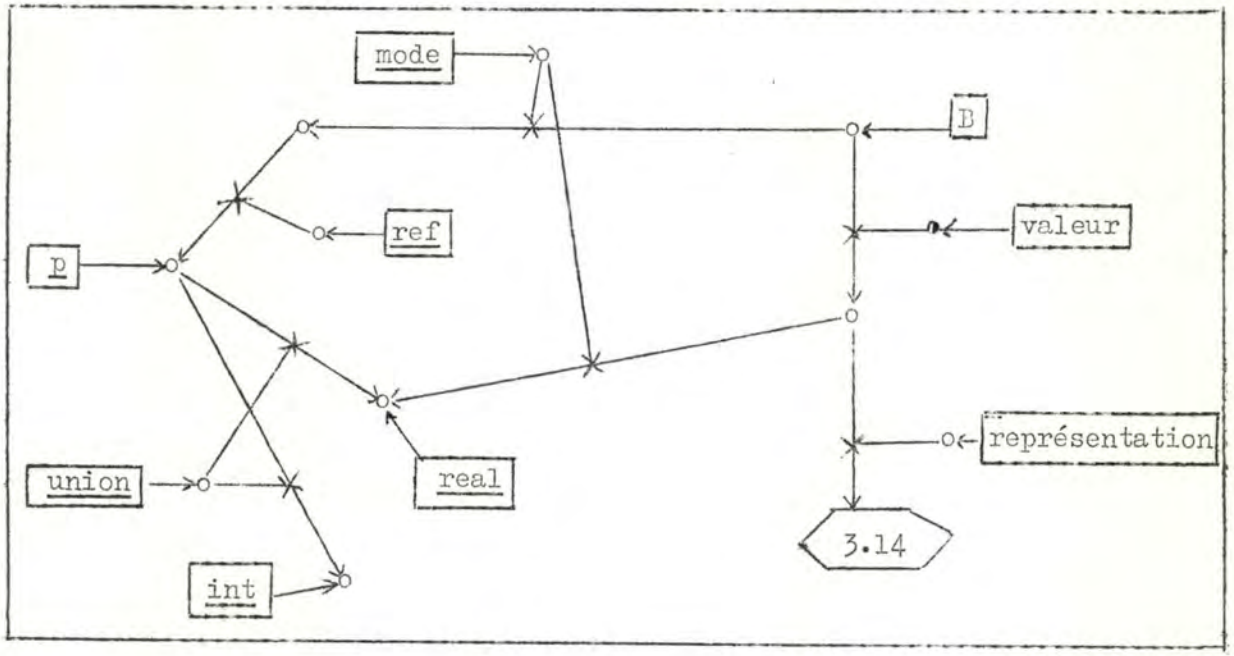
ref real x ; x := 3.14 ;

En effet, en accord avec le concept habituel de variable, il nous semble normal de considérer qu'une variable a un attribut de nature "valeur" et que la valeur de cet attribut a un attribut "représentation". Dans cette façon de voir, les diagrammes B.3 et B.5 seraient modifiés par le remplacement de l'attribut "valeur" par l'attribut "représentation". Quant aux diagrammes B.2 et B.4, ils nous donneraient respectivement B.7 et B.8.



B.7

ref ref ref ref real Y



B.8

mode p = union (real, int) ; ref p B ; B := 3.14 ;

C. Différences et similitudes entre Opérandes et Opérateurs

En Algol 60 comme en Algol 68, la distinction entre opérateurs et opérandes est nette et explicite.
Nous nous proposons d'analyser dans ce paragraphe les raisons d'une telle distinction et de voir si les similitudes existant malgré tout entre ces deux groupes d'objets ne permettent pas de prévoir que, dans un langage plus extensible que ceux dont nous disposons à ce jour, les opérateurs et opérandes ne formeront plus qu'un seul groupe.

En Algol 60, les opérateurs n'ont pas de mode, car les routines n'y sont pas des valeurs. Il est donc impossible de les utiliser pour un but autre que celui défini par le langage. Ce ne sont que des outils servant à manipuler les opérandes et rien d'autre. Il en est de même dans un grand nombre de langages.

En Algol 68, par contre, les opérateurs ont un mode et les routines forment un domaine de base, ce sont des valeurs. On peut définir des variables auxquelles il est possible d'assigner des routines. Un premier pas a donc été franchi vers la fusion des deux groupes.

Cependant, a priori, tout comme Strachey distinguait des objets de première ou seconde classe en fonction de leur comportement par rapport à l'assignation, -comportement voulu par les auteurs du langage -, en Algol 68, on peut distinguer les opérandes des opérateurs car un objet donné ne peut en aucun cas être à la fois l'un et l'autre. Ceci est dû à l'absence d'opérateurs ayant pour opérandes des objets de mode routine. Cette absence est, elle aussi, voulue par les auteurs du langage. La raison principale de cette décision des auteurs d'Algol 68 est qu'ils désirent que la compilation des programmes soit possible et que les programmeurs soient protégés contre des erreurs dangereuses pour l'exécution du programme.

Mais si l'on considère la notion d'opérateur en son sens le plus général, les opérations dont nous avons parlé en A, les coercitions et autres fonctions primitives sont donc des opérateurs.
A ce moment, pour autant que l'on se place avant la compilation, nous

avons déjà à notre disposition un certain nombre d'objets qui jouent tantôt le rôle d'opérande, tantôt celui d'opérateur : les "déclarateurs". En effet, dans la déclaration

mode zut = struct (real a, int b) par exemple,

zut joue le rôle d'opérande, mais si maintenant nous déclarons

zut a ; zut joue cette fois-ci le rôle d'opérateur (dont le rôle est d'établir une relation "désigne" entre le nom a et une notion définie par un attribut de nature mode et de valeur zut.

La distinction provient donc du fait que l'effet des opérateurs ayant pour arguments des objets pouvant être à leur tour des opérateurs peut être défini a priori (lors de la compilation) et de façon statique.

On peut concevoir, à notre avis, qu'un langage extensible réponde aux conditions suivantes :

- tous les objets de base ont un mode, y compris les routines;
- il existe des opérateurs définis sur tous les modes, y compris sur les routines;
- il est possible de définir de nouveaux objets et modes à partir de ceux de base.

Dans un tel langage, les opérateurs pourraient donc être traités tantôt comme opérateurs, tantôt comme opérandes. Cela permettrait par exemple de définir le produit de composition de fonctions, etc...

Il ne nous semble donc plus nécessaire, dans un langage de ce genre, d'établir une distinction réelle entre opérateurs et opérandes, puisque tous deux finissent par jouer le même rôle.

Chapitre Cinq

CLASSIFICATION ET CONCLUSIONS

Notre objectif principal, dans ce dernier chapitre, est d'établir une classification des notions nommées et/ou nommables dans les langages algorithmiques en général.

A. Classification des notions

Tout comme dans les classifications étudiées aux chapitres deux et trois, ici encore nous allons regrouper les notions en domaines de base et dérivés.

1. Domaines de base

Si, pour un langage donné, on essaie de découvrir quelles en sont les notions de base, en se basant sur l'idée intuitive qu'une notion de base est une notion que le programmeur ne peut pas définir lui-même, on se trouve confronté à un problème de choix. En effet, supposons que le langage en question définisse les entiers et les réels et permette de définir des structures par exemple. Il est alors possible de concevoir qu'un programmeur définisse les réels sous forme de structures à deux éléments entiers, mantisse et exposant.

Par une telle définition, le programmeur obtient en fait non pas les réels eux-mêmes mais une représentation de réels en termes d'entiers.

Dans ce cas, les réels sont-ils de base ?

S'il est possible d'en définir une représentation, avoir les réels comme domaine de base peut être redondant.

Le problème revient donc à savoir comment on désire utiliser les réels et en fonction de cela, décider si on les considère comme faisant partie des domaines de base ou pas.

Cette question du choix nous a amené à poser la définition suivante :

> Un domaine est de base si et seulement si les notions qui le composent sont définies et non définissables.

Dans le cas particulier où le domaine est composé d'une et une seule notion, on dira que celle-ci est de base. On a donc :

> La condition nécessaire et suffisante pour qu'un mode soit de base est qu'il soit défini et non définissable.

En effet, si un mode est défini et non définissable, il est impossible à un programmeur de le définir et il est donc de base. Inversément, s'il est de base, il est défini et non définissable car s'il était définissable, il ne pourrait être de base et s'il n'était pas défini, il faudrait le définir pour qu'il existe et il ne serait donc pas de base non plus.

Quels sont les domaines de base ? Il est bien évident que ces domaines ne sont pas toujours les mêmes pour chaque langage et qu'il ne serait donc pas utile de les énumérer. Nous préférons en donner quelques caractéristiques.

Considérons tout d'abord les notions qui ne sont pas des modes. Toutes les notions appartenant à un domaine de base par rapport à un langage (modes non compris) ont un attribut de nature "représentation" dont la valeur est un objet de l'univers considéré par ce langage (voir chapitre IV, paragraphe B).

On peut ensuite séparer les domaines de base en deux groupes distincts, ceux pour lesquels il existe un attribut de nature mode, et ceux pour lesquels il n'en existe pas.

Il est assez intéressant de voir que, en général, les modes correspondant aux domaines de base sont des modes de base.

Un autre critère permet de regrouper ces domaines : la nommabilité. Nous avons ainsi quatre groupes, par ordre croissant d'extensibilité :

- les notions qui ne sont ni nommées ni nommables;
- les notions nommées;
- les notions nommées et nommables;
- les notions nommables.

La façon habituelle de classer les domaines de base en fonction du "genre" de leurs éléments est encore valable ici.
On obtient les trois groupes suivants :

- données exemples : réels, entiers, ...
- opérations exemples : opérateurs, fonctions primitives, routines
- auxiliaires exemples : formats, labels.

Il est évident que l'on peut grouper les domaines de base suivant tous les critères mentionnés ci-dessus.

2. Domaines Dérivés

Tout comme pour les domaines de base, on peut séparer les domaines dérivés en fonction de l'existence ou de l'absence de l'attribut mode pour les notions qui les composent.

Nous allons regrouper les domaines dérivés en différentes classes suivant les "propriétés" des notions qui les composent. Par "propriété" d'une notion, nous entendons la structure d'attribut y afférente .

Une première classe est celle des notions qui possèdent un attribut "représentation" dont la valeur est un objet de l'univers dont parle le langage considéré. Il s'agit principalement de domaines assimilés (au sens donné au chapitre trois) aux domaines de base et à des domaines comme "scalar" ou "subrange" (Pascal).

Une deuxième classe regroupe les notions qui possèdent un attribut "valeur" dont la valeur est une notion et dont la valeur de l'attribut "mode" possède un attribut "référence" (pointeurs et variables).
(voir diagrammes IV B.6 et 7)

Une troisième classe est formée du domaine "union" (voir Algol 68). Les notions de ce domaine ont un attribut "valeur", mais la valeur de cet attribut est une notion dont le mode (la valeur de l'attribut mode) n'est défini qu'à l'exécution, à un instant donné. (diagramme IV.B.8)

Quatre autres classes seront encore définies, en fonction de la composition des notions y appartenant.

- nombre fixe d'éléments de même mode (tableaux à bornes fixes);
- nombre variable d'éléments de même mode (tableaux à bornes variables, files, "class" (Pascal), "powerset" (Pascal)...);
- nombre fixe d'éléments de modes différents (structures, listes, "trees",...);
- nombre variable d'éléments de modes différents ("queues", chaînes..)

Une dernière classe reprendrait les labels au cas où ceux-ci seraient dérivés et non repris dans une classe déjà citée.

Un dernier critère de classification, et le plus important dans le cadre de ce travail, est celui de la nommabilité.
Trois cas, et non plus quatre comme pour les domaines de base, sont possibles :

- nommés
- nommables
- nommés et nommables.

A l'intérieur d'un langage il n'existe pas de notions définissables qui ne soient ni nommées ni nommables.

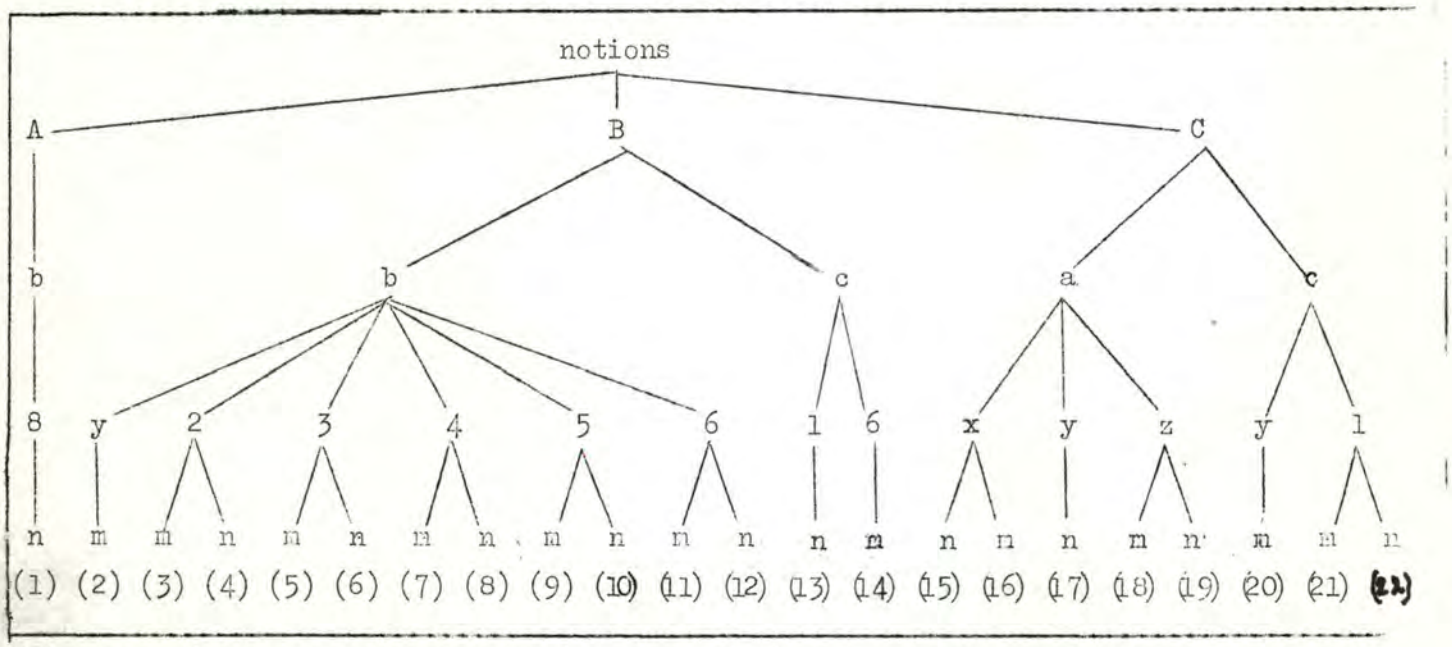
3. Classification des notions nommées et nommables

Si l'on désire classifier les notions sous l'angle de la nommabilité, on obtient trois grands groupes : les notions nommées, les notions nommables et les notions nommées et nommables.
Chacun de ces groupes se subdivise en sous-groupes : les sous-groupes des notions définies et non définissables (appartiennent à des domaines de base ou sont des modes de base), des notions définissables et des notions définies et définissables (ces deux derniers groupes sont dérivés).
Le premier de ces trois sous-groupes se décompose en "données", "opérations" et "auxiliaires", tandis que les deux derniers se décomposent en les huit classes citées au paragraphe 2.

Une nouvelle subdivision nous donne les notions qui sont des modes et celles qui n'en sont pas.

Pour un langage particulier, il est évident que certains sous-groupes n'existent pas. L'ensemble de ces différentes possibilités forme un arbre non binaire. Pour Algol 68, cet arbre est représenté au diagramme A.1.
Nous y avons pris les conventions suivantes :

A signifie nommées, B nommables, C nommées et nommables, a définies b définissables, c définies et définissables, x données, y opérations, z auxiliaires, 1 à 8 les huit classes dans l'ordre cité au paragraphe 2, m modes et n non-modes.



(1) labels (supposés définissables)
(2) modes routines
(3) modes références
(4) notions références
(5) modes unions
(6) notions unions
(7) modes } multiples à bornes fixes
(8) notions }
(9) modes } multiples à bornes variables
(10) notions }
(11) modes } structures
(12) notions }
(13) notions bytes, sema et compl.
(14) modes sema, compl, bytes, file
(15) notions domaines de base "données"
(16) modes de base
(17) notions routines
(18) mode format
(19) notions formats
(20) mode routine proc
(21) modes bits, string
(22) notions bits, string

B. Conclusions

Nous avions pour but principal de classifier les êtres que les langages algorithmiques nomment ou permettent de nommer par un mot.
Nous désirions aussi examiner quelles étaient les extensions possibles et tenir compte des notions de mode et de représentation.
Les différentes approches que nous avons suivies nous ont-elles permis d'atteindre ce but, telle est la question qui se pose maintenant.

La classification de Strachey est limitée aux objets qui ne sont pas des opérateurs, et il n'y est pas fait mention explicite des modes, pas plus que des représentations. Elle nous a permis, entre autres, de faire une nette distinction entre objets nommés et objets nommables.
Il nous semble que cette approche est surtout adaptée à des langages simples, bien "cloisonnés", c'est-à-dire où les différents domaines sont bien distincts. En effet, en Pascal, par exemple, D contient L, Id et V↖(E*) et les domaines n'y sont donc plus distincts. Un diagramme du genre II.B.2 se réduit presque à D tout seul, ce qui n'est plus très significatif.
Pour des langages extensibles et non "cloisonnés", la méthode de Strachey nous paraît difficilement applicable. Ainsi que nous l'avons déjà soulevé au chapitre deux (C.3), les outils d'extension utilisés dans cette approche nous semblent nettement insuffisants et laissent, à notre avis, trop de place à l'arbitraire. En particulier, la notation "X → Y" signifie en pratique tout aussi bien n'importe quelle fonction de X dans Y que certaines fonctions ou une fonction bien précise.
Nos essais d'adaptation de cette méthode aux opérateurs et aux modes n'ont pas abouti à des résultats satisfaisants à notre point de vue.

En fait, cette approche est essentiellement non constructive et c'est la raison pour laquelle plus le langage auquel on tentera de l'appliquer sera extensible et donc complexe et général, moins les "cloisons" y seront apparentes ou même présentes, et donc moins la schématisation sera significative. Il est toujours difficile, en effet, d'appliquer une méthode de formalisation non constructive à un sujet complexe sans perdre la plus grande partie de sa signification première.

Algol 68, par contre se prête bien à une classification complète et précise, à l'exception toutefois des labels à propos desquels le langage indique très peu de choses du point de vue de leur sémantique. Les modes et les opérateurs y sont pris en considération.
Pour établir cette classification, nous nous sommes inspiré du Rapport. En effet, pour Algol 68, comme pour PL/1, une méthode de description formalisée du langage a été définie. Ce sont des méthodes constructives, cependant le formalisme utilisé est relativement lourd et très long à déchiffrer.

Il nous semble qu'une méthode de formalisation de ce genre doit non seulement être constructive mais contenir en elle-même les moyens de la faire évoluer. En effet, si l'on emploie une méthode constructive pour décrire un langage de façon formelle, il ne faut pas que la complexité du langage entraîne une trop grande complexité des "outils" de construction. Il faut cependant remarquer que, si le formalisme employé par A. Van Wijngaarden dans la définition d'Algol 68 répond à ce critère, il reste malgré tout assez aride.

Parmi les méthodes constructives auto-évolutives, nous avons abordé au chapitre quatre celle des trois-graphes (notion, nature d'attribut, valeur d'attribut) développée par C. Cherton. C'est cette méthode, combinée avec Algol 68, qui a servi de base à la classification proposée au début de ce chapitre.

Une autre piste semble pouvoir être intéressante : l'approche de Ph. Jorrand (voir Bibliographie). Cette approche, qui utilise aussi une schématisation à partir de graphes, propose de définir un langage sans aucun type de base mais fournissant des mécanismes pour en définir. Les mécanismes d'extension permettraient aussi de définir des relations entre ces types (exemple : $\in$, $\cup$, $\cap$, $\subset$, x), des conversions d'un type à l'autre avec leurs niveaux et des procédures. Cette optique s'attache spécialement à mettre en valeur la notion de représentation.

CLASSIFICATION D'ALGOL 60 PAR STRACHEY

1. Domaines de Base

T booléens
N entiers
R réels
L locations
Q strings
J jump points
S états de la mémoire

2. Domaines Dérivés

$E = D + V$ valeurs des expressions

$V = T + R + N$ objets stockables

$D = L + L^{*} + J + Q + P + W + W^{*}$ objets nommables

avec $P = [D^{*} \longrightarrow [S \rightarrow S]] + [D^{*} \longrightarrow [S \rightarrow V \times S]]$ fonctions et procédures

et $W = [S \longrightarrow E \times S]$ les calls by name

W^{*} désigne les switches.

Nous avons ci-dessus reproduit fidèlement la classification donnée par Strachey. Dans le texte cependant, nous nous sommes permis d'y apporter quelques modifications, en ce qui concerne Q et les switches. Voir chapitre II, paragraphe A.2.2.

-=-=-=-=-=-=-=-

SYNTAXE DE BCPL

1. Expressions E

E ::= <name> | <stringconst> | <charconst> | <number> |
true | false | (E) | valof C | lv E | rv E |
E(<E list>) | E() |
E * E | E / E | E rem E |
E + E | E - E | + E | - E |
E = E | E ≠ E | E ls E | E gr E | E le E | E ge E |
E lshift E | E rshift E |
not E |
E ∧ E |
E ∨ E |
E ≡ E | E ≢ E |
E → E, E | table <constant> { , <constant> }

<E list> ::= E { , E }

<constant> ::= E

2. Commandes C

C ::= <E list> := <E list> | E(<E list>) | E() | goto E |
<name> : C | if E do C | while E do C | unless E do C |
until E do C | C repeat | C repeatuntil E |
C repeatwhile E | test E then C or C | break | return |
finish | resultis E | for <name> = E to E do C |
for <name> = E to E by <constant> do C |
switchon E into C | case <constant> : C | endcase |
default : C | <block> | <compound command> | <empty>

3. Déclarations

D : : = <name> (<FPL>) = E | <name> (<FPL>) be C |
<name list> = <E list> | <name> = vec <constant>

<FPL> : : = <name list> | <empty>

<name list> : : = <name> { , <name> }

<block> : : = $(<declaration> { <declaration> } { ; C } $)

<compound command> : : = $(C { ; C } $)

<declaration> : : = let D { and D } | static <decl body> |
manifest <decl body> | global <decl body>

<decl body> : : = $(<C def> { ; <C def> } $)

<C def> : : = <name> : <constant> | <name> = <constant>

<program> : : = <declaration> { <declaration> } { ; C } |
C { ; C }

4. On suppose prédéfinies les catégories suivantes :

<number> : : = <digit> $\left\{ \text{<digit>} \right\}_{0}^{\infty}$

<digit> : : = 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0

<letter> : : = a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r |
s | t | u | v | w | x | y | z | A | B | C | D | E | F | G | H | I | J |
K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

<special symbol> : : = * | " | ' | = | ≠ | < | > | ...

<string const> : : = " $\left\{ \text{<string character>} \right\}_{0}^{\infty}$ "

<charconst> : : = ' < string character > '

<string character> : : = <letter> | <digit> | <special symbol>
| <empty>

<name> : : = <letter> $\left\{ \text{<string character>} \right\}_{0}^{\infty}$

<empty> représente le symbole "blanc".

Note : { } indique la répétition.
Cette syntaxe comporte des ambiguïtés qui sont résolues dans le manuel BCPL (voir bibliographie) de M. Richards.

SYNTAXE DE PASCAL

1. Notation, terminologie et vocabulaire

<letter> ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R |
S | T | U | V | W | X | Y | Z | a | b | c | d | e | f | g | h | i | j |
k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<special symbol> ::= + | - | * | / | ∨ | ∧ | ¬ | = | ≠ | < | > | ≤ | ≥ | (|) |
[|] | { | } | := | $_{10}$ | . | , | ; | : | ' | ↑ | div |
mod | nil | in | if | then | else | case | of | repeat |
until | while | do | for | to | downto | begin | end |
with | goto | var | type | array | record | powerset |
file | class | function | procedure | const

2. Identificateurs et nombres

<identifier> ::= <letter> <letter or digit>*

<letter or digit> ::= <letter> | <digit>

<number> ::= <integer> | <real number>

<integer> ::= <digit>$^{\oplus}$

<real number> ::= <digit>$^{\oplus}$. <digit>$^{\oplus}$ |
<digit>$^{\oplus}$. <digit>$^{\oplus}$ $_{10}$ <scale factor> | <integer> $_{10}$
<scale factor>

<scale factor> ::= <digit>$^{\oplus}$ | <sign> <digit>$^{\oplus}$

<sign> ::= + | -

3. Définitions de constantes

<unsigned constant> ::= <number> | ' <character>$^{\oplus}$ ' |
<identifier> | nil

<constant> ::= <unsigned constant> | <sign> <number>

<constant definition> ::= <identifier> = <constant>

Note : {...} indique la répétition, $\oplus$ 1 ou plusieurs fois et
$*$ o ou plusieurs fois.

4. Définitions de types de données

<type> ::= <scalar type> | <subrange type> | <array type> |
<record type> | <powerset type> | <file type> |
<class type> | <pointer type> | <type identifier>

<type identifier> ::= <identifier>

<type definition> ::= <identifier> = <type>

a) <scalar type> ::= (<identifier> { , <identifier> }*)

standard scalar type :

integer

real

boolean (false, true)

char

alfa (string de longueur n)

<subrange type> ::= <constant> .. <constant>

b) <array type> ::= array [<index type> { , <index type> }*]
of <component type>

<component type> ::= <type>

<index type> ::= <scalar type> | <subrange type> | <type identifier>

<record type> ::= record <field list> end

<field list> ::= <fixed part> | <fixed part> ; <variant part> |
<variant part>

<fixed part> ::= <record section> { ; <record section> }*

<record section> ::= <field identifier> { , <field identifier> }*
: <type>

<variant part> ::= case <tag field> : <type identifier>
of <variant> { ; <variant> }*

<variant> ::= { <case label> : }* (<field list>) |
{ <case label> }+

<case label> ::= <unsigned constant>

<tag field> ::= <identifier>

<field identifier> ::= <identifier>

c) < powerset type > : : = powerset < type identifier > | powerset < sub-range type >

d) < file type > : : = file of < type >

e) < class type > : : = class < max num > of < type >
< max mum > : : = < integer >

f) < pointer type > : : = ↑ < class variable >
< class variable > : : = < variable >

5. Déclarations et dénotations de variables

< variable declaration > : : = < identifier > { , < identifier > }* : < type >
< variable > : : = < entire variable > | < component variable >
< entiro variable > : : = < variable identifier >
< variable identifier > : : = < identifier >
< component variable > : : = < indexed variable > | < field designator > |
< current file component > | < referenced component >

a) < indexed variable > : : = < array variable > [< expression > { , < expression > }*]
< array variable > : : = < variable >

b) < field designator > : : = < record variable > . < field identifier >
< record variable > : : = < variable >

c) < current file component > : : = < file variable > ↑
< file variable > : : = < variable >

d) < referenced component > : : = < pointer variable > ↑
< pointer variable > : : = < variable >

6. Expressions

< factor > : : = < variable > | < unsigned constant > | < function designator > | < set > | (< expression >) | ¬ < factor >
< set > : : = [< expression > { , < expression > }*] | [ ]
< term > : : = < factor > | < term > < multiplying operator > < factor >

⟨simple expression⟩ :: = ⟨term⟩ |
⟨simple expression⟩ ⟨adding operator⟩ ⟨term⟩ |
⟨adding operator⟩ ⟨term⟩

⟨expression⟩ :: = ⟨simple expression⟩ |
⟨simple expression⟩ ⟨relational operator⟩
⟨simple expression⟩

⟨multiplying operator⟩ :: = * | / | div | mod | ∧

⟨adding operator⟩ :: = + | - | ∨

⟨relational operator⟩ :: = = | ≠ | < | ≤ | ≥ | > | in

⟨function designator⟩ :: = ⟨function identifier⟩
(⟨actual parameter⟩ { , ⟨actual parameter⟩ }*)

⟨function identifier⟩ :: = ⟨identifier⟩

7. Instructions

⟨statement⟩ :: = ⟨simple statement⟩ | ⟨structured statement⟩

a) ⟨simple statement⟩ :: = ⟨assignment statement⟩ | ⟨procedure statement⟩ | ⟨goto statement⟩

⟨assignment statement⟩ :: = ⟨variable⟩ := ⟨expression⟩ |
⟨function identifier⟩ := ⟨expression⟩

⟨procedure statement⟩ :: = ⟨procedure identifier⟩ | ⟨procedure identifier⟩ (⟨actual parameter⟩ { , ⟨actual parameter⟩ }*)

⟨procedure identifier⟩ :: = ⟨identifier⟩

⟨actual parameter⟩ :: = ⟨expression⟩ | ⟨variable⟩ | ⟨procedure identifier⟩ | ⟨function identifier⟩

⟨goto statement⟩ :: = goto ⟨label⟩

⟨label⟩ :: = ⟨integer⟩

b) ⟨structured statement⟩ :: = ⟨compound statement⟩ | ⟨conditional statement⟩ | ⟨repetitive statement⟩ ⟨with statement⟩

⟨compound statement⟩ :: = begin
⟨component statement⟩ { ; ⟨component statement⟩ }* end

< component statement > : : = < statement > | < label definition > < statement >

< label definition > : : = < label > :

< conditional statement > : : = < if statement > | < case statement >

< if statement > : : = if < expression > then < statement > |
if < expression > then < statement > else < statement >
(expressions booléennes)

< case statement > : : = case < expression > of < case list element > { ; < case list element > }* end

< case list element > : : = { < case label > : }⊕ < statement > |
{ < case label > : }⊕

< repetitive statement > : : = < while statement > |
< repeat statement > | < for statement >

< while statement > : : = while < expression > do < statement >
(expression booléenne)

< repeat statement > : : = repeat < statement > { ; < statement > }* until < expression >
(expression booléenne)

< for statement > : : = for < control variable > := < for list > do < statement >

< for list > : : = < initial value > to < final value > | < initial value > downto < final value >

< control variable > : : = < identifier >

< initial value > : : = < expression >

< final value > : : = < expression >

< with statement > : : = with < record variable > do < statement >

8. Déclarations de procédure

< procedure declaration > : : = < procedure heading >
< constant definition part > < type definition part >
< variable declaration part > < procedure and function declaration part > < statement part >

a) < procedure heading > :: = procedure < identifier > ; |
procedure < identifier > (< formal parameter section >
{; < formal parameter section >}*);
< formal parameter section > :: =
< parameter group > |
const < parameter group > {; < parameter group >}* |
var < parameter group > {; < parameter group >}* |
function < parameter group > |
procedure < identifier > {, < identifier >}*
< parameter group > :: = < identifier > {, < identifier >}*
: < type identifier >

b) < constant definition part > :: = < empty > |
const < constant definition > {, < constant definition >}*;

c) < type definition part > :: = < empty > |
type < type definition > {; < type definition >}*;

d) < variable declaration part > :: = < empty > |
var < variable declaration > {; < variable declaration >}*;

e) < procedure and function declaration part > :: = {< procedure or declaration part > ;}*
< procedure or declaration part > :: = < procedure declaration > |
< function declaration >

f) < statement part > :: = < compound statement >

9. Déclarations de fonctions

< function declaration > :: = < function heading >
< constant definition part > < type definition part >
< variable declaration part > < procedure and function declaration part > < statement part >

< function heading > :: = function < identifier >
(< formal parameter section > {; < formal parameter section >}*):
< result type > ;
< result type > :: = < type identifier >

10. Programme

< program > : : = < constant definition part > < type definition part >
< variable declaration part > < procedure and function part >
< statement part > .

-=-=-=-=-=-=-=-

CLASSIFICATION DE BCPL ETENDUE AUX OPERATEURS

Domaines de Base

"opérandes" {
- LV locations
- RV binary-bit-patterns ($\mathcal{B}_b$-valeurs)
- S états de la mémoire
- J jump points

opérateurs {
- AD opérateurs arithmétiques dyadiques
- AM opérateurs arithmétiques monadiques
- R opérateurs relationnels
- SH opérateurs de shift
- LD opérateurs logiques dyadiques
- LM opérateurs logiques monadiques
- Te opérateurs de test
- lv opérateur lv
- rv opérateur rv
- Sél opérateur de sélection

Rappelons que, sont de base les opérateurs définis et non définissables. Cela n'empêche pas que l'on puisse les exprimer en termes de fonctions entre domaines d'"opérandes", bien que ce ne soit pas nécessaire. On a ainsi, en supposant défini $T = \{ \underline{true}, \underline{false} \} = \{1...1, 0...0\} \subset RV$ et N comme le domaine des entiers (ayant un code appartenant à RV) :

$AD = [N \times N \longrightarrow N] = \{+, -, *, /, \underline{rem}\}$

$AM = [N \longrightarrow N] = \{+, -\}$

$R = [RV \times RV \longrightarrow T] = \{=, \neq, \underline{ls}, \underline{gr}, \underline{le}, \underline{ge}\}$

$SH = [RV \times N_+ \longrightarrow RV] = \{\underline{lshift}, \underline{rshift}\}$

N_+ est l'ensemble des entiers positifs.

$LD = [T \times T \longrightarrow T] = \{\wedge, \vee, =, \neq\}$

$LM = [T \rightarrow T] = \{\underline{not}\}$

$Te = T \times S \dashrightarrow E$ (E voir domaines dérivés)

$\underline{lv} = E \times S \dashrightarrow [LV \longrightarrow RV]$

$\underline{rv} = E \times S \dashrightarrow [RV \longrightarrow RV \times S]$

$Sel = Pt \times S \times N_+ \longrightarrow LV \times S$

On remarquera que la notation $X \longrightarrow Y$ laisse supposer "une quelconque fonction" définie sur X et à valeurs dans Y. En fait, ici il s'agit bien de quelques fonctions bien particulières. La notation $X \rightarrow Y$ est en fait très imprécise.

Domaines dérivés

D = LV variables, paramètres formels (appelés par valeur)
+ RV constantes, case labels
+ F fonctions, routines
+ J jump points
+ Pt vecteurs, tables
+ Q strings

V = LV (opérateurs $\underline{lv}$ et $\underline{rv}$)
+ RV constantes

E = D + V valeurs des expressions

$F = [E^* \longrightarrow [S \dashrightarrow RV \times S]]$ fonctions

$+ [E^* \longrightarrow [S \rightarrow S]]$ routines.

$Pt = [LV \times S \dashrightarrow (LV^*) \times S]$

$Q = [LV \times S \dashrightarrow (LV^*) \times S]$

-=-=-=-=-=-=-=-=-

CLASSIFICATION DE PASCAL ETENDUE AUX MODES

Domaines de Base

S états de la mémoire

L locations

J jump points

T booléens

R réels

N entiers

C caractères

C^n "alfa" (n est un paramètre de l'implémentation)

Id identificateurs

Domaines Dérivés

$V = Sc + Pt + E^*$ constantes, pointeurs et "sets"

$E = D' + V = D' + Sc + E^*$ valeurs des expressions

on a $Sc = T + N + R + C + C^n + Id$ constantes

et $D' = L + A + Re + CFL + Pt + Pw$ (voir D) variables

$F = \left[(D' + F)^* \longrightarrow [S \to S]\right]$ procédures

$+ \left[(D' + F)^* \longrightarrow [S \longrightarrow (Sc + Pt) \times S]\right]$ fonctions

$W = [S \longrightarrow (D' + F) \times S]$ paramètres formels appelés par nom

$W_1 = \left[D' \times S \longrightarrow [(Sc \setminus R\_) \longrightarrow D' \times S]\right]$ "variant part" dans les records.

Domaine des choses nommables D

$D = T + N + R + C + C^n + Id$ constantes

D' variables
- $+ L$ variables de type scalaire, standard, subrange
- $+ A$ variables de type array
- $+ Re$ variables de type record
- $+ CFL$ variables de type class et file
- $+ Pt$ variables de type pointer
- $+ Pw$ variables de type powerset

(suite page suivante)

+ F	procédures et fonctions
+ W	calls by name
+ W_1	variant part dans les records
+ J	jump points

D" modes

+ $\mathbb{N}$ + $\mathbb{T}$ + $\mathbb{R}$ + $\mathbb{C}$ + $\mathbb{C}^n$	modes standard
+ Id*	modes scalaires
+ Sb	modes subranges
+ Ar	modes arrays
+ MRe	modes records
+ MCFL	modes class et files
+ MPt	modes pointers
+ MPw	modes powersets

Détaillons les domaines qui sont encore imprécis :

$Re = (D' + W_1)^*$

$Pt = [L \times S \longrightarrow A \times S]$

$Pw = [S \longrightarrow (L^*) \times S]$

$A = L^* + CFL^* + Pt^* + Re^* + Pw^* + A^*$

$MPt = [S \longrightarrow A \times S]$ (donc similaire à CFL)

$MRe = (Sc + D'' + W_2)^*$

avec $W_2 = [(Sc + D'') \times S \longrightarrow [(Sc \setminus R) \longrightarrow (Sc + D'') \times S]]$

pour les parties variables.

Définissons maintenant

$\mathcal{D} = \{ T, N, R, C, C^n, Id^*, L, A, CFL, Pt, Pw, Re, Sb, Ar, MRe, MCFL, MPt, MPw \}$

(ici T, N,... sont considérés comme des éléments, donc un domaine est considéré comme un tout!)

et $\mathcal{S}c = \{ T, N, R, C, C^n, Id^* \}$

on a alors

$$Ar = \underset{X \in \mathcal{D}}{+} (X\divideontimes^{*})$$

$$MCFL = \underset{X \in \mathcal{D}}{+} [S \longrightarrow (X\divideontimes) \times S]$$

$$MPw = \underset{Y \in \mathcal{J}c}{+} \mathcal{P}^{\circ}(Y) \qquad (\mathcal{P} \text{ ensemble des parties})$$

$$Sb = \underset{Y \in \mathcal{J}c}{+} Y\divideontimes \qquad \text{(il manque la notion d'ordre)}$$

La complexité de ces derniers domaines nous a amené à introduire de nouvelles notations.

B I B L I O G R A P H I E

J.C. BOUSSARD et C. PAIR — "Introduction à Algol 68" Revue d'Informatique et de Recherche Opérationnelle B. 3, 1969.

M.J. BOWLDEN — "Macros in Higher-Level Languages" A.C.M. vol. 6, n° 12, dec. 1971, Sigplan Notices Proceeding of the international Symposium on Extensible Languages.

P. BRANQUART, J. LEWI, M. SINTZOFF, Q.P.L. WODON — "The Composition of Semantics in Algol 68" C.A.C.M., nov. 1971, vol. 14, n° 11.

J.N. BUXTON et autres — "C.P.L. elementary programming manual", Univ. Computer Lab., Cambridge, 1966.

Cl. CHERTON, M. MEWITT, D. FRANK, F. DOYEN — "Essai de formalisation de la sémantique", en cours de publication.

Cl. CHERTON — "Sur une méthode de définition formelle des langages algorithmiques" Thèse de doctorat, 1970.

F. DUNCAN — "The ultimate Meta-Language", extrait de "Formal Language Description Languages for Computer Programming", 1966.

FOX — "An assesment of Algol 68", Infotech 1972.

D. GRIES — "Compiler Construction for Digital computer", Wiley, 1971.

M. GROSS, Q.A. LENTIN — "Notions sur les grammaires formelles", Gauthier-Villars, Paris, 1967.

B. HIGMAN — "A comparative Study of Programming Languages" Mc Donald, 1967.

M.E. HOPKINS — "A case for the GOTO" A.C.M., août 1972, Boston, Proceedings of Annual Conference.

Ph. JORRAND — "Data types and Extensible languages" A.C.M. vol 6, n° 12, déc. 1971, Sigplan Notices, Proceedings of the International Symposium on Extensible Languages.

Ph. JORRAND, S.A. SCHUMAN — "Définition d'un langage ou définition d'une implémentation ?", AFCET, Congrès Grenoble 1972, Les Techniques de l'Informatique.

Ph. JORRAND, S.A. SCHUMAN — "Definition mechanisms in extensible programming languages", Centre scientifique de Grenoble, Etude, FF 2.0111.

Ph. JORRAND, D. BERT — "On some Basic Concepts for Extensible Programming Languages", Proceedings of the International Computing Symposium, Venise, avril 1972.

Ph. JORRAND — "Remarques sur les langages extensibles", AFCET, Congrès d'Informatique, Paris, sept. 1970.

Ph. JORRAND — "Some aspects of BASEL, the base language for an extensible language facility", Sigplan Notices, vol 4, n° 8, août 1969, Proceedings of the Extensible Languages Symposium.

Ph. JORRAND — "Tutorial on Algol 68", Proceedings on the third annual Princeton Conference on Information Sciences and Systems, mars 1969.

D.E. KNUTH "The Art of Computer Programming"
(3 vol) Addison-Wesley, Reading (Mass.), 1968.

B.M. LEAVENWORTH "Programming With(out) the GOTO"
ACM Proceedings on the Annual Conference, août 1972,
Boston.

C.H. LINDSEY "Algol 68 with fewer tears"
Computer Journal, vol 15 n° 2.

C.H. LINDSEY, "Informal Introduction to Algol 68".
S.G. VAN DER MEULEN IFIP North Holland Publ. Company, 1971.

Mc CARTHY "Lisp 1.5. Programmer's Manual",
M.I.T. Press.

M. NAUR, éd. "Revised Report on the Algorithmic Language Algol 60"
CACM n° 6, 1963.

R.J. NELSON "Introduction to Automata"
John Wiley and Sons, 1968.

D. PARK "Some Semantics for Data Structures",
Progr. Research Group, University of Oxford.

M. RICHARDS "BCPL : A tool for Compiler writing and system programming"
Proc. AFIPS vol 34, Spring Joint Computer Conference,
1969.

M. RICHARDS "The BCPL reference Manual"
Memo 69/1 University Computer Lab., Cambridge 1969.

S. ROSEN "Programming Systems and Languages"
Mc Graw Hill, 1967.

RUTISHAUER "Descriptions of Algol 60"
Springer Verlag, 1967.

J.E. SAMMET "Programming Languages : History and Fundamentals"
Prentice-Hall Inc., 1969.

D. SCOTT & Ch. STRACHEY — "Toward a Mathematical Semantics for Computer Languages" Proceedings of the Symposium on Computers and Automata. M.R.I. Symposia series, vol 21., Polytechnic Institute of Brooklyn, 1971.

T.B. STEEL ed. — "Formal Language Description Languages for Computer Programming" Proceedings of the I.F.I.P. working Conference on formal Language Description Languages, 1966.

Ch. STRACHEY — "Fundamental Concepts in Programming Languages" Cours donné à l'Université d'Oxford.

Ch. STRACHEY — "Toward a formal Semantics" Extrait de STEEL éd., 1966.

Ch. STRACHEY — "Varieties of Programming Languages" Infotech 1972.

Ch. STRACHEY — "Varieties of Programming Languages" International Computing Symposium, Venice 12-14 avril 1972.

T. VAN GILS — "Syntactic Definition Mechanisms" ACM vol 6, n° 12, déc. 1971, Sigplan Notices, Proceedings of the International Symposium on Extensible Languages.

A. VAN WIJNGAARDEN — "Report on the Algorithmic Language Algol 68" Springer-Verlag, 1969.

P. WEGNER — "Programming Languages, Information Structures and Machine Organization" Computer Science Series, Mc Graw Hill, 1968.

N. WIRTH & C.A.R. HOARE — "A contribution to the development of Algol" CACM, juin 1966.

N. WIRTH — "The Programming Language PASCAL" Acta Informatica 1, Springer Verlag, 1971.

W.A. WULF — "A case against the Goto" ACM, août 1972, Boston, Proceedings Annual Conference.

— o —